

34th European Conference on Object-Oriented Programming

ECOOP 2020, November 15–17, 2020, Berlin, Germany
(Virtual Conference)

Edited by

Robert Hirschfeld

Tobias Pape



Editors

Robert Hirschfeld 

Hasso Plattner Institute, University of Potsdam, Germany
robert.hirschfeld@hpi.uni-potsdam.de

Tobias Pape 

Hasso Plattner Institute, University of Potsdam, Germany
tobias.pape@hpi.uni-potsdam.de

ACM Classification 2012

Software and its engineering

ISBN 978-3-95977-154-2

Published online and open access by

Schloss Dagstuhl – Leibniz-Zentrum für Informatik GmbH, Dagstuhl Publishing, Saarbrücken/Wadern, Germany. Online available at <https://www.dagstuhl.de/dagpub/978-3-95977-154-2>.

Publication date

November, 2020

Bibliographic information published by the Deutsche Nationalbibliothek

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

License

This work is licensed under a Creative Commons Attribution 3.0 Unported license (CC-BY 3.0):
<https://creativecommons.org/licenses/by/3.0/legalcode>.



In brief, this license authorizes each and everybody to share (to copy, distribute and transmit) the work under the following conditions, without impairing or restricting the authors' moral rights:

- Attribution: The work must be attributed to its authors.

The copyright is retained by the corresponding authors.

Digital Object Identifier: 10.4230/LIPIcs.ECOOP.2020.0

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.

Editorial Board

- Luca Aceto (*Chair*, Gran Sasso Science Institute and Reykjavik University)
- Christel Baier (TU Dresden)
- Mikolaj Bojanczyk (University of Warsaw)
- Roberto Di Cosmo (INRIA and University Paris Diderot)
- Javier Esparza (TU München)
- Meena Mahajan (Institute of Mathematical Sciences)
- Dieter van Melkebeek (University of Wisconsin-Madison)
- Anca Muscholl (University Bordeaux)
- Luke Ong (University of Oxford)
- Catuscia Palamidessi (INRIA)
- Thomas Schwentick (TU Dortmund)
- Raimund Seidel (Saarland University and Schloss Dagstuhl – Leibniz-Zentrum für Informatik)

ISSN 1868-8969

<https://www.dagstuhl.de/lipics>

■ Contents

Preface	
<i>Robert Hirschfeld</i>	0:ix
Message from the General Chair	
<i>Christian Hammer</i>	0:xi–0:xii
Message from the Artifact Evaluation Chairs	
<i>Lisa Nguyen Quang Do and Manuel Rigger</i>	0:xiii–0:xiv
Objects and a Changing World: Foreword by the President of AITO	
<i>Eric Jul</i>	0:xv
Organization	
.....	0:xvii–0:xviii
List of Authors	
.....	0:xix–0:xxi
List of Reviewers	
.....	0:xxiii–0:xxvii

Regular Papers

Sound Regular Corecursion in coFJ	
<i>Davide Ancona, Pietro Barbieri, Francesco Dagnino, and Elena Zucca</i>	1:1–1:28
Perfect Is the Enemy of Good: Best-Effort Program Synthesis	
<i>Hila Peleg and Nadia Polikarpova</i>	2:1–2:30
Blame for Null	
<i>Abel Nieto, Marianna Rapoport, Gregor Richards, and Ondřej Lhoták</i>	3:1–3:28
Static Race Detection and Mutex Safety and Liveness for Go Programs	
<i>Julia Gabet and Nobuko Yoshida</i>	4:1–4:30
Reconciling Event Structures with Modern Multiprocessors	
<i>Evgenii Moiseenko, Anton Podkopaev, Ori Lahav, Orestis Melkonian, and Viktor Vafeiadis</i>	5:1–5:26
Don't Panic! Better, Fewer, Syntax Errors for LR Parsers	
<i>Lukas Diekmann and Laurence Tratt</i>	6:1–6:32
K-LLVM: A Relatively Complete Semantics of LLVM IR	
<i>Liyi Li and Elsa L. Gunter</i>	7:1–7:29
Space-Efficient Gradual Typing in Coercion-Passing Style	
<i>Yuya Tsuda, Atsushi Igarashi, and Tomoya Tabuchi</i>	8:1–8:29
Multiparty Session Programming With Global Protocol Combinators	
<i>Keigo Imai, Rumyana Neykova, Nobuko Yoshida, and Shoji Yuen</i>	9:1–9:30



Designing with Static Capabilities and Effects: Use, Mention, and Invariants (Pearl) <i>Colin S. Gordon</i>	10:1–10:25
Owicki-Gries Reasoning for C11 RAR <i>Sadegh Dalvandi, Simon Doherty, Brijesh Dongol, and Heike Wehrheim</i>	11:1–11:26
A Semantics for the Essence of React <i>Magnus Madsen, Ondřej Lhoták, and Frank Tip</i>	12:1–12:26
Test-Case Reduction via Test-Case Generation: Insights from the Hypothesis Reducer (Tool Insights Paper) <i>David R. MacIver and Alastair F. Donaldson</i>	13:1–13:27
Model-View-Update-Communicate: Session Types Meet the Elm Architecture <i>Simon Fowler</i>	14:1–14:28
Static Analysis of Shape in TensorFlow Programs <i>Sifis Lagouvardos, Julian Dolby, Neville Grech, Anastasios Antoniadis, and Yannis Smaragdakis</i>	15:1–15:29
Value Partitioning: A Lightweight Approach to Relational Static Analysis for JavaScript <i>Benjamin Barslev Nielsen and Anders Møller</i>	16:1–16:28
Static Type Analysis by Abstract Interpretation of Python Programs <i>Raphaël Monat, Abdelraouf Ouadjaout, and Antoine Miné</i>	17:1–17:29
Reference Mutability for DOT <i>Vlastimil Dort and Ondřej Lhoták</i>	18:1–18:28
Tackling the Awkward Squad for Reactive Programming: The Actor-Reactor Model <i>Sam Van den Vonder, Thierry Renaux, Bjarno Oeyen, Joeri De Koster, and Wolfgang De Meuter</i>	19:1–19:29
A Framework for Resource Dependent EDSLs in a Dependently Typed Language (Pearl) <i>Jan de Muijnck-Hughes, Edwin Brady, and Wim Vanderbauwhede</i>	20:1–20:31
Data Consistency in Transactional Storage Systems: A Centralised Semantics <i>Shale Xiong, Andrea Cerone, Azalea Raad, and Philippa Gardner</i>	21:1–21:31
Putting Randomized Compiler Testing into Production (Experience Report) <i>Alastair F. Donaldson, Hugues Evrard, and Paul Thomson</i>	22:1–22:29
Lifting Sequential Effects to Control Operators <i>Colin S. Gordon</i>	23:1–23:30
Flow-Sensitive Type-Based Heap Cloning <i>Mohamad Barbar, Yulei Sui, and Shiping Chen</i>	24:1–24:26
Scala with Explicit Nulls <i>Abel Nieto, Yaoyu Zhao, Ondřej Lhoták, Angela Chang, and Justin Pu</i>	25:1–25:26
A Type-Directed Operational Semantics For a Calculus with a Merge Operator <i>Xuejing Huang and Bruno C. d. S. Oliveira</i>	26:1–26:32

Row and Bounded Polymorphism via Disjoint Polymorphism <i>Ningning Xie, Bruno C. d. S. Oliveira, Xuan Bi, and Tom Schrijvers</i>	27:1–27:30
A Trusted Infrastructure for Symbolic Analysis of Event-Driven Web Applications <i>Gabriela Sampaio, José Fragoso Santos, Petar Maksimović, and Philippa Gardner</i>	28:1–28:29
The Duality of Subtyping <i>Bruno C. d. S. Oliveira, Cui Shaobo, and Baber Rehman</i>	29:1–29:29
Safe, Flexible Aliasing with Deferred Borrows <i>Chris Fallin</i>	30:1–30:26

Abstracts of “Science of Computer Programming” Journal-first Papers

Reshape Your Layouts, Not Your Programs: A Safe Language Extension for Better Cache Locality <i>Alexandros Tasos, Juliana Franco, Sophia Drossopoulou, Tobias Wrigstad, and Susan Eisenbach</i>	31:1–31:3
A Big Step from Finite to Infinite Computations <i>Davide Ancona, Francesco Dagnino, Jurriaan Rot, and Elena Zucca</i>	32:1–32:2
Abstracting Gradual References <i>Matías Toro and Éric Tanter</i>	33:1–33:4

■ Preface

ECOOP is a conference about programming. Originally its focus was on object orientation, but now it looks at a much broader range of programming topics. Areas of interest include the design, implementation, optimization, analysis, and theory of programs, programming languages, and programming environments. The conference welcomes innovative and creative solutions to real problems, evaluations that provide new insights into existing solutions, and reproduction studies.

This year ECOOP received 71 submissions categorized by their authors as research papers, tool insight papers, reproduction studies, experience reports, pearls, or brave new ideas. Papers were evaluated based on originality, significance, evidence, and clarity. After careful and thorough review following a light double-blind, identify-the-champion process, 30 of them were accepted by the Program and External Review Committees. Papers written by committee members received extra reviews by the rest of the committee. While some papers received up to six reviews, none had fewer than three.

Due to the COVID-19 pandemic, a physical meeting of the Program Committee was impossible. Despite that, all members of the Program and the External Review Committees did exceptional work, dealt with every obstacle in their way, and stayed positive and constructive, leading to rich and interesting proceedings.

ECOOP 2020 was planned to be held in July in Berlin, Germany, but went virtual and co-located with ACM SIGPLAN's SPLASH conference. Together with OOPSLA, Onward!, GPCE, SLE, DLS, SAS, and several workshops, ECOOP took place in November 2020.

The 2020 AITO Dahl-Nygaard Junior Prize was awarded to Jonathan Bell for his significant contributions to tooling in the Java ecosystem, which has improved our understanding and ability to test and discover bugs in software. The Senior Prize was presented to Jan Vitek, whose work has been to observe how software is being developed and how programming languages are being used; over his career he has studied and improved practical programming languages.

A journal-first arrangement with Elsevier's Science of Computer Programming yielded the first "Special Issue on Selected Papers from the European Conference on Object-Oriented Programming."

It was an honor and a privilege to serve as Program Chair for this edition of ECOOP. I would like to thank the following: all authors who submitted their research; my amazing colleagues of the Program and External Review Committees along with our other external reviewers for their outstanding work and help; the Artifact Evaluation Committee; the Organizing Committee; and my Software Architecture Group here at HPI for their invaluable support.

Robert Hirschfeld
Fall 2020



■ Message from the General Chair

It is my great pleasure to welcome you to ECOOP'20, to be held during 15–17 Nov, the first virtual instance of ECOOP in its 34 years of history. ECOOP is the European forum for bringing together researchers, practitioners, and students to share their ideas and experiences on all topics related to programming languages, software development, object-oriented technologies, systems and applications.

The Corona pandemic quickly raised doubts whether we could have a physical meeting in Berlin during July, as originally planned. Like many other conferences we had to evaluate the situation carefully and eventually came to the conclusion that a physical meeting, while not forbidden by law in the German state of Berlin at the planned timeframe, would not be possible, mostly due to international travel restrictions.

Instead, ECOOP co-located with SPLASH, which gave the organizing team time to plan a virtual meeting and the authors their online presentation. Other conferences and several of ECOOP's satellite events like the Scala Symposium followed, such that SPLASH this year is a true multi-conference on programming and related communities. This year's ECOOP features a doctoral symposium and a poster session, which are jointly organized with SPLASH, as well as tutorials. In the joint SPLASH virtualization team we developed a plan for ECOOP/SPLASH to be experienced world-wide, in particular with closed captioning and a daily 12h program that is repeated in the subsequent half day. Experience was also drawn from other virtually organized conferences like PLDI, ICSE, and ICFP. While nothing can compensate for the networking at a physical meeting, we therefore hope that we can provide you with the best online experience possible.

My congratulations go to the junior and senior AITO Dahl-Nygaard Prize winners, Jonathan Bell and Jan Vitek, who will present keynotes during ECOOP, and the paper “Load-Time Structural Reflection in Java” by Shigeru Chiba, which was selected for the AITO Test of Time Award.

Organizing ECOOP in these challenging times would not have been possible without the support of a great team. I would like to express my gratitude towards all the people involved in organizing this year's ECOOP and the joint virtual event, in particular the members of the ECOOP'20 Organizing Committee, the joint SPLASH Virtualization Committee, especially the intersection of those two consisting of Toni Mattis, Patrick Rein, and Fabio Niephaus, the Program Committee led by Robert Hirschfeld for compiling an excellent program, AITO e.V. and the support from its Executive Board. Many people contributed to various aspects of the program: the Doctoral Symposium was chaired by Philipp Dominik Schubert, Nafise Eskandani Masoule, Chengsong Tan; Julia Belyakova served as Diversity Chair; Annabel Satin served as Finance Chair; Jan Vitek and Gregor Richards co-organized the workshops; Jacob Hughes and Toni Mattis served as Student Volunteer Co-Chairs; Tim Felgentreff and Tobias Pape managed the ECOOP web site; Fabio Niephaus served as Publicity Chair; Eric Bodden served as Sponsorship Chair; Goran Piskachev and Patrik Rein served as Posters Co-Chairs; and last but not least, Lisa Nguyen Quang Do and Manuel Rigger served as Artifact Evaluation Co-Chairs for ECOOP 2020.

I would like to gratefully acknowledge our sponsor AITO and funding by Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) – HA 6869/2-1, our financial supporters Facebook, Google, Connex Communications GmbH, and Huawei Technologies, as well as the cooperation with ACM and SIGPLAN.

0:xii Message from the General Chair

Finally, I wish all attendants of ECOOP'20 and the co-located events a fantastic time, thought-provoking, and inspiring talks, stimulating discussions and that the online conference offers ample opportunities to network with your peers, researchers and practitioners from our vibrant community.

Christian Hammer
Fall 2020

■ Message from the Artifact Evaluation Chairs

The goal of the *Artifact Evaluation* (AE) is to foster the reproducibility of results by providing authors the possibility to submit an artifact for accepted papers. For ECOOP 2020, artifacts include, but are not limited to, software artifacts, data sets, and proofs. An *Artifact Evaluation Committee* (AEC) reviews these artifacts and decides upon their acceptance. The accepted artifacts are archived in the *Dagstuhl Artifacts Series* (DARTS) published on the *Dagstuhl Research Online Publication Server* (DROPS). Each artifact is assigned a *Digital Object Identifier* (DOI) that can be used in future citations.

This year, the committee evaluated 21 artifacts out of 29 papers accepted at the conference's research track. This corresponds to a participation rate of 72%. 20 of those artifacts were accepted, marking a 95% acceptance rate.

In total 70% of the research papers published at ECOOP 2020 have successfully passed the AE process, indicated by an artifact-evaluation badge on the paper. This is an improvement over the previous ECOOP editions: from 2017 to 2019, respectively 59%, 38%, and 50% of the research papers were accompanied by accepted artifacts.

The AE process for 2020 was a continuation of the AE process of previous ECOOP editions. In particular, the process was based on the artifact evaluation guidelines by Shriram Krishnamurthi, Matthias Hauswirth, Steve Blackburn, and Jan Vitek published on the Artifact Evaluation site.¹ In addition, the authors and reviewers were provided with guidelines for creating and reviewing software artifacts, in particular guidelines from the Artifact Evaluation site,² the *HOWTO for AEC Submitters* by Dan Barowy, Charlie Curtsinger, Emma Tosch, and John Vilk,³ Marianna Rapoport's *Proof Artifacts – Guidelines for Submission and Reviewing*,⁴ and Erin Dahlgren's study on the OOPSLA 2019 artifact evaluation process.

Each artifact was evaluated by three AEC members, which corresponded to a reviewer load of two to three artifacts. The reviewing process consisted of three phases:

- In the *kick-the-tires* phase, reviewers briefly verified the basic integrity of the artifacts to discover any issues that could prevent the evaluation of the artifact (e.g., a corrupted virtual machine image) and to assign a grade for the getting-started guide.
- In case of any issues, reviewers could, during the *interactive reviewing period*, indicate issues and ask clarifying questions to the authors. Authors, in turn, could respond to the reviewers' feedback, and update their artifacts to answer questions and address issues that the reviewers could then also respond to. During that phase, reviewers started a more comprehensive evaluation of their assigned artifacts. They were asked to assess the consistency of the artifact with respect to the paper, the artifact's completeness, documentation, and reusability for future research and to decide on an overall grade.
- In the *final reviewing period*, the submission system was closed to the authors. Each reviewer had a week to finish the evaluation of their assigned artifacts.

The review phase was then followed by a discussion phase, in which artifacts were discussed to converge on either the artifacts' acceptance or rejection.

¹ <http://www.artifact-eval.org>

² <https://www.artifact-eval.org/guidelines.html>

³ <http://bit.ly/HOWTO-AEC>

⁴ <https://proofartifacts.github.io/guidelines/>

Authors that received an acceptance notification were given one week of time to incorporate reviewers' feedback and submit the camera-ready version of their artifacts.

We would like to commend the efforts of all 23 members of this year's AEC, who, in spite of the global crisis, donated their valuable time and effort to make the AE process possible. We thank Martin Kavalari, Philipp Markovics, and Jan Vitek for their efforts in enabling Nextjournal as an option for authors to submit and host their artifacts. We would also like to thank Michael Wagner and the DARTS team for their efforts enabling the publication of the artifacts volume, as well as ECOOP 2020's General Chair Christian Hammer, and the Program Chair Robert Hirschfeld for helping us coordinate the AE with the paper review process.

Lisa Nguyen Quang Do and Manuel Rigger
Fall 2020

■ Objects and a Changing World

Foreword by the President of AITO

The world has changed abruptly: The arrival of a new corona virus has seriously impacted society and traditional academic conferences have not been spared, but rather cancelled, or, at best, become virtual as ECOOP 2020. And, already, we have to start seriously considering going virtual in 2021, thus possibly missing out two years in a row, both Berlin 2020 and Aarhus 2021. We may also already now start wondering about the Post-Corona virus time that we hope will follow the likely (we hope) approval and adoption of a corona vaccine: What will that look like? Will we be able to go back to the tradition physical-meeting-for-a-week academic conferences? Will we want to? Or will everyone, including funding agencies, find that, well, virtual is good enough and a lot cheaper? I hope not as the personal interactions at physical conferences is very hard to mimic virtually. What will happen, time will tell.

Another change that has been slowly evolving over the past decades is that the original strong Object Orientation of ECOOP is that OO has become mainstream, and has diversified: thus ECOOP is mellowing into a more general programming language conference.

I would like to thank the ECOOP 2020 Program Chair, his crew and his PC for putting together an excellent program, Christian Hammer and his crew for organizing ECOOP 2020, and Annabel Satin who has been AITO's indispensable behind-the-scene-manager. I hope you-all will enjoy ECOOP 2020.

Sincerely,
Eric Jul



■ Organization

General Chair

Christian Hammer (*University of Potsdam, Germany*)

Program Chair

Robert Hirschfeld (*Hasso Plattner Institute, University of Potsdam, Germany*)

Workshops Co-Chairs

Gregor Richards (*University of Waterloo, Canada*)

Jan Vitek (*Northeastern University, USA*)

Student Volunteer Co-Chairs

Jacob Hughes (*King's College London, United Kingdom*)

Toni Mattis (*Hasso Plattner Institute, University of Potsdam, Germany*)

Virtualization Co-Chairs

Toni Mattis (*Hasso Plattner Institute, University of Potsdam, Germany*)

Patrick Rein (*Hasso Plattner Institute, University of Potsdam, Germany*)

Fabio Niephaus (*Hasso Plattner Institute, University of Potsdam, Germany*)

Artifact Evaluation Co-Chairs

Lisa Nguyen Quang Do (*Google, Switzerland*)

Manuel Rigger (*ETH Zurich, Switzerland*)

Posters Co-Chairs

Goran Piskachev (*Fraunhofer IEM, Germany*)

Patrick Rein (*Hasso Plattner Institute, University of Potsdam, Germany*)

Diversity Chair

Julia Belyakova (*Northeastern University, USA*)

Sponsorship Chair

Eric Bodden (*Heinz Nixdorf Institut, Paderborn University and Fraunhofer IEM, Germany*)

Finance Chair

Annabel Satin (*P.C.K., United Kingdom*)

34th European Conference on Object-Oriented Programming (ECOOP 2020).

Editors: Robert Hirschfeld and Tobias Pape



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



Doctoral Symposium Co-Chairs

Philipp Dominik Schubert (*Heinz Nixdorf Institut, Paderborn University, Germany*)

Nafise Eskandani Masoule (*Technical University of Darmstadt, Germany*)

Chengsong Tan (*King's College London, United Kingdom*)

Publicity Chair

Fabio Niephaus (*Hasso Plattner Institute, University of Potsdam, Germany*)

Web Technology Co-Chairs


Tim Felgentreff (*Oracle Labs, Potsdam, Germany*)

Tobias Pape (*Hasso Plattner Institute, University of Potsdam, Germany*)

Publications Chair

Tobias Pape (*Hasso Plattner Institute, University of Potsdam, Germany*)

■ List of Authors


Davide Ancona  (1, 32)
DIBRIS, University of Genova, Italy

Anastasios Antoniadis (15)
University of Athens, Greece

Mohamad Barbar (24)
University of Technology Sydney, Australia;
CSIRO's Data61, Sydney, Australia

Pietro Barbieri  (1)
DIBRIS, University of Genova, Italy


Xuan Bi (27)
The University of Hong Kong, China

Edwin Brady  (20)
University of St Andrews, United Kingdom

Andrea Cerone (21)
Department of Computing,
Imperial College London, United Kingdom


Angela Chang (25)
University of Waterloo, Canada

Shiping Chen (24)
CSIRO's Data61, Sydney, Australia

Francesco Dagnino  (1, 32)
DIBRIS, University of Genova, Italy

Sadegh Dalvandi  (11)
University of Surrey, United Kingdom

Joeri De Koster  (19)
Software Languages Lab,
Vrije Universiteit Brussel, Belgium


Wolfgang De Meuter  (19)
Software Languages Lab,
Vrije Universiteit Brussel, Belgium


Jan de Muijnck-Hughes  (20)
University of Glasgow, United Kingdom


Lukas Diekmann (6)
Software Development Team,
King's College London, United Kingdom

Simon Doherty  (11)
University of Sheffield, United Kingdom

Julian Dolby (15)
IBM Research, Yorktown Heights, NY, USA

Alastair F. Donaldson  (13, 22)
Google, London, United Kingdom;
Imperial College London, United Kingdom

Brijesh Dongol  (11)
University of Surrey, United Kingdom


Vlastimil Dort  (18)
Charles University, Prague, Czech Republic

Sophia Drossopoulou (31)
Imperial College London, United Kingdom;
Microsoft Research, London, United Kingdom

Susan Eisenbach (31)
Imperial College London, United Kingdom


Hugues Evrard (22)
Google, London, United Kingdom

Chris Fallin (30)
Mozilla, Mountain View, CA, USA


Simon Fowler  (14)
University of Edinburgh, United Kingdom

José Fragoso Santos (28)
INESC-ID/Instituto Superior Técnico,
Universidade de Lisboa, Portugal;
Imperial College London, United Kingdom

Juliana Franco (31)
Microsoft Research, London, United Kingdom

Julia Gabet  (4)
Imperial College London, United Kingdom


Philippa Gardner (21, 28)
Department of Computing,
Imperial College London, United Kingdom

Colin S. Gordon  (10, 23)
Department of Computer Science,
Drexel University, Philadelphia, PA, USA

Neville Grech (15)
University of Athens, Greece

Elsa L. Gunter (7)
Department of Computer Science,
University of Illinois at Urbana-Champaign,
Urbana, IL, USA

Xuejing Huang  (26)
The University of Hong Kong, China


Atsushi Igarashi  (8)
Graduate School of Informatics, Kyoto
University, Japan

Keigo Imai  (9)
Gifu University, Japan


- Sifis Lagouvardos (15)
University of Athens, Greece
- Ori Lahav (5)
Tel Aviv University, Israel
- Ondřej Lhoták  (3, 12, 18, 25)
University of Waterloo, Canada
- Liyi Li (7)
Department of Computer Science,
University of Illinois at Urbana-Champaign,
Urbana, IL, USA
- David R. MacIver  (13)
Imperial College London, United Kingdom
- Magnus Madsen  (12)
Aarhus University, Denmark
- Petar Maksimović (28)
Imperial College London, United Kingdom
- Orestis Melkonian (5)
University of Edinburgh, UK
- Antoine Miné  (17)
Sorbonne Université, CNRS, LIP6, Paris, France;
Institut Universitaire de France, Paris, France
- Evgenii Moiseenko (5)
St. Petersburg State University, Russia;
JetBrains Research, St. Petersburg, Russia
- Raphaël Monat  (17)
Sorbonne Université, CNRS, LIP6, Paris, France
- Anders Møller  (16)
Aarhus University, Denmark
- Rumyana Neykova  (9)
Brunel University London, United Kingdom
- Benjamin Barslev Nielsen (16)
Aarhus University, Denmark
- Abel Nieto  (3, 25)
University of Waterloo, Canada
- Bjarno Oeyen  (19)
Software Languages Lab,
Vrije Universiteit Brussel, Belgium
- Bruno C. d. S. Oliveira (26, 27, 29)
The University of Hong Kong, China
- Abdelraouf Ouadjaout  (17)
Sorbonne Université, CNRS, LIP6, Paris, France
- Hila Peleg  (2)
University of California San Diego, CA, USA
- Anton Podkopaev (5)
National Research University Higher School of
Economics, Moscow, Russia;
MPI-SWS, Kaiserslautern, Germany;
JetBrains Research, St. Petersburg, Russia
- Nadia Polikarpova  (2)
University of California San Diego, CA, USA
- Justin Pu (25)
University of Waterloo, Canada
- Azalea Raad (21)
MPI-SWS, Kaiserslautern, Germany
- Marianna Rapoport (3)
University of Waterloo, Canada
- Baber Rehman (29)
The University of Hong Kong, China
- Thierry Renaux  (19)
Software Languages Lab, Vrije Universiteit
Brussel, Belgium
- Gregor Richards  (3)
University of Waterloo, Canada
- Jurriaan Rot (32)
Radboud University, The Netherlands
- Gabriela Sampaio (28)
Imperial College London, United Kingdom
- Tom Schrijvers  (27)
KU Leuven, Belgium
- Cui Shaobo (29)
University of California San Diego, CA, USA
- Yannis Smaragdakis (15)
University of Athens, Greece
- Yulei Sui (24)
University of Technology Sydney, Australia
- Tomoya Tabuchi (8)
Graduate School of Informatics, Kyoto
University, Japan
- Éric Tanter (33)
PLEIAD Laboratory, Computer Science
Department (DCC), University of Chile,
Santiago, Chile
- Alexandros Tasos (31)
Imperial College London, United Kingdom
- Paul Thomson (22)
Google, London, United Kingdom
- Frank Tip  (12)
Northeastern University, Boston, MA, USA

Matías Toro (33)

PLEIAD Laboratory, Computer Science
Department (DCC), University of Chile,
Santiago, Chile

Laurence Tratt  (6)

Software Development Team,
King's College London, United Kingdom

Yuya Tsuda  (8)


Graduate School of Informatics,
Kyoto University, Japan

Viktor Vafeiadis (5)

MPI-SWS, Kaiserslautern, Germany

Sam Van den Vonder  (19)

Software Languages Lab,
Vrije Universiteit Brussel, Belgium

Wim Vanderbauwhede  (20)

University of Glasgow, United Kingdom

Heike Wehrheim  (11)

Paderborn University, Germany

Tobias Wrigstad (31)

Uppsala University, Sweden

Ningning Xie (27)


The University of Hong Kong, China

Shale Xiong (21)

Department of Computing,
Imperial College London, United Kingdom

Nobuko Yoshida  (4, 9)

Imperial College London, United Kingdom

Shoji Yuen  (9)

Nagoya University, Japan

Yaoyu Zhao (25)

University of Waterloo, Canada

Elena Zucca  (1, 32)

DIBRIS, University of Genova, Italy

■ List of Reviewers

Program Committee

Robert Hirschfeld (*chair*)
Hasso Plattner Institute
University of Potsdam
Germany
robert.hirschfeld@gmx.net

Karim Ali
University of Alberta
USA
karim.ali@ualberta.ca

Davide Ancona
DIBRIS, University of Genova
Italy
davide.ancona@unige.it

Carl Friedrich Bolz-Tereick
Heinrich-Heine-Universität Düsseldorf
Germany
cfbolz@gmx.de

John Boyland
Univeristy of Wisconsin
USA
boyland@cs.uwm.edu

Shigeru Chiba
The University of Tokyo
Japan
chiba@chibas.net

Theo D'Hondt
Vrije Universiteit Brussel
Belgium
tjdhondt@vub.ac.be

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

Sebastian Erdweg
JGU Mainz
Germany
erdweg@uni-mainz.de

Tim Felgentreff
Oracle Labs, Potsdam
Germany
tim.felgentreff@hpi.uni-potsdam.de

Olivier Flückiger
Northeastern University
USA
olivf@ccs.neu.edu

Lidia Fuentes
Universidad de Málaga
Spain
lff@lcc.uma.es

Richard P. Gabriel
Dream Songs, Inc. & HPI
California & Germany
rpg@dreamsongs.com

Anitha Gollamudi
Harvard University
USA
agollamudi@g.harvard.edu

Elisa Gonzalez Boix
Vrije Universiteit Brussel
Belgium
egonzale@vub.ac.be

Philipp Haller
KTH Royal Institute of Technology
Sweden
hallerp@gmail.com

Christian Hammer
University of Potsdam
Germany
hammer@cs.uni-potsdam.de

Felienne Hermans
Leiden University
The Netherlands
f.f.j.hermans@liacs.leidenuniv.nl

Atsushi Igarashi
Kyoto University
Japan
igarashi@kuis.kyoto-u.ac.jp



Stephen Kell
University of Kent
United Kingdom
stephen.kell@cl.cam.ac.uk

Raffi Khatchadourian
City University of New York (CUNY)
Hunter College
USA
raffi.khatchadourian@hunter.cuny.edu

Yu David Liu
State University of New York (SUNY)
Binghamton
USA
davidl@cs.binghamton.edu

Hidehiko Masuhara
Tokyo Institute of Technology
Japan
masuhara@acm.org

James Noble
Victoria University of Wellington
New Zealand
kjsx@ecs.vuw.ac.nz

Klaus Ostermann
University of Tübingen
Germany
klaus.ostermann@uni-tuebingen.de

Patrick Rein
Hasso Plattner Institute
University of Potsdam
Germany
patrick.rein@hpi.uni-potsdam.de

Guido Salvaneschi
University of St. Gallen
Switzerland
guido.salva@gmail.com

Manuel Serrano
Inria
France
manuel.serrano@inria.fr

Jeremy G. Siek
Indiana University
USA
jsiek@indiana.edu

Friedrich Steimann
Fernuniversität Hagen
Germany
steimann@fernuni-hagen.de

Emma Söderberg
Lund University
Sweden
emma.m.soderberg@gmail.com

Peter Thiemann
University of Freiburg
Germany
thiemann@acm.org

Eli Tilevich
Virginia Tech
USA
tilevich@cs.vt.edu

Frank Tip
Northeastern University
USA
f.tip@northeastern.edu

Jan Vitek
Northeastern University
USA
j.vitek@neu.edu

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

Tijs van der Storm
CWI & University of Groningen
The Netherlands
storm@cwi.nl

External Reviewer Committee

Robert Hirschfeld (*chair*)
Hasso Plattner Institute
University of Potsdam
Germany
robert.hirschfeld@gmx.net

Erik Ernst
Google
Denmark
ernst@acm.org

Matthew Flatt
University of Utah
 USA
 mflatt@cs.utah.edu

Jeremy Gibbons
Department of Computer Science
University of Oxford
United Kingdom
 jeremy.gibbons@cs.ox.ac.uk

Doug Lea
State University of New York (SUNY)
Oswego
 USA
 dl@cs.oswego.edu

Crista Lopes
University of California, Irvine
California
 lopes@uci.edu

Toni Mattis
Hasso Plattner Institute
University of Potsdam
Germany
 toni.mattis@hpi.de

Todd Millstein
University of California, Los Angeles
 USA
 todd@cs.ucla.edu

Jens Palsberg
University of California, Los Angeles
 USA
 palsberg@ucla.edu

Tomas Petricek
University of Kent
United Kingdom
 tomas@tomasp.net

Benjamin C. Pierce
University of Pennsylvania
 USA
 bcpierce@cis.upenn.edu

Joe Gibbs Politz
University of California, San Diego
 USA
 jpolitz@cs.swarthmore.edu

Tiark Rompf
Purdue University
 USA
 tiark@purdue.edu

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

Additional Reviewers

Leonidas Lampropoulos
University of Pennsylvania
University of Maryland
 USA
 leonidaslamp@hotmail.com

Magnus Madsen
University of Waterloo
Canada
 magnusm@cs.au.dk

Matthias Springer
Google
Japan
 me@matthiasspringer.de

Nicholas Rioux
University of Pennsylvania
 USA
 nrioux@seas.upenn.edu

Rui Abreu
University of Lisbon
Portugal
 rui.maranhao@tecnico.ulisboa.pt

Artifact Evaluation Committee

Lisa Nguyen Quang Do (*chair*)
Google
Switzerland
 lisa.nqd@gmail.com

Manuel Rigger (*chair*)
ETH Zurich
Switzerland
 manuel.rigger@inf.ethz.ch

Ellen Arteca
Northeastern University
USA
earteca@uwaterloo.ca

Alexandre Bartel
University of Luxembourg
alexandre.bartel@uni.lu

Francesco Dagnino
DIBRIS, University of Genova
Italy
francesco.dagnino@dibris.unige.it

Erin Dahlgren
University of Chicago
USA
edahlgren@uchicago.edu

Kiko Fernandez-Reyes
Uppsala University
Sweden
kiko.fernandez@it.uu.se

Zheng Guo
University of California, San Diego
USA
zhg069@ucsd.edu

Byron Hawkins
INRIA Rennes
France
byron.hawkins@inria.fr

Pinjia He
ETH Zurich
Switzerland
pinjia.he@inf.ethz.ch

Aarti Kashyap
University of British Columbia
Canada
kaarti.sr@gmail.com

Alyssa Milburn
Vrije Universiteit Amsterdam
The Netherlands
a.a.milburn@vu.nl

Felix Pauck
Paderborn University
Germany
fpauck@mail.uni-paderborn.de

Cedric Richter
Paderborn University
Germany
cedricr@mail.upb.de

Andrea Rosà
University of Lugano
Switzerland
andrea.rosa@usi.ch

Somesh Singh
Indian Institute of Technology, Madras
India
somesh.singh1992@gmail.com

Justin Smith
Lafayette College, Easton
USA
justinssmith1@gmail.com

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

Qiyi Tang
University of Oxford
United Kingdom
qiyi.tang71@gmail.com

John Toman
University of Washington, Seattle
USA
jtoman@cs.washington.edu

Alix Trieu
Aarhus University
Denmark
alix.trieu@cs.au.dk

Alexi Turcotte
Northeastern University
USA
aturcotte@uwaterloo.ca

Junwen Yang
University of Chicago
USA
junwen@uchicago.edu

Chengyu Zhang
East China Normal University
China
dale.chengyu.zhang@gmail.com

Reviewers

0:xxvii

Fuyuan Zhang
MPI-SWS
Germany
fuyuan@mpi-sws.org

Daming Zou
Peking University
China
zoudm@pku.edu.cn

Sound Regular Corecursion in coFJ

Davide Ancona 


DIBRIS, University of Genova, Italy
davide.ancona@unige.it

Pietro Barbieri 

DIBRIS, University of Genova, Italy
pietro.barbieri@edu.unige.it

Francesco Dagnino 

DIBRIS, University of Genova, Italy
francesco.dagnino@dibris.unige.it

Elena Zucca 

DIBRIS, University of Genova, Italy
elena.zucca@unige.it

Abstract

The aim of the paper is to provide solid foundations for a programming paradigm natively supporting the creation and manipulation of cyclic data structures. To this end, we describe COFJ, a Java-like calculus where objects can be *infinite* and methods are equipped with a *codefinition* (an alternative body). We provide an abstract semantics of the calculus based on the framework of *inference systems with corules*. In COFJ with this semantics, FJ recursive methods on finite objects can be extended to infinite objects as well, and behave as desired by the programmer, by specifying a codefinition. We also describe an operational semantics which can be directly implemented in a programming language, and prove the soundness of such semantics with respect to the abstract one.

2012 ACM Subject Classification Theory of computation → Operational semantics; Software and its engineering → Recursion; Software and its engineering → Semantics

Keywords and phrases Operational semantics, coinduction, programming paradigms, regular terms

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2020.1

Related Version A full version of the paper is available at <https://arxiv.org/abs/2005.14085>.

Funding *Davide Ancona*: Member of GNCS (Gruppo Nazionale per il Calcolo Scientifico), INdAM (Istituto Nazionale di Alta Matematica “F. Severi”)

Introduction

Applications often deal with data structures which are conceptually infinite, such as streams or infinite trees. Thus, a major problem for programming languages is how to finitely represent something which is infinite, and, even harder, how to correctly manipulate such finite representations to reflect the expected behaviour on the infinite structure.

A well-established solution is *lazy evaluation*, as, e.g., in Haskell. In this approach, the conceptually infinite structure is represented as the result of a function call, which is evaluated only as much as needed. Focusing on the paradigmatic example of streams (infinite lists) of integers, we can define `two_one = 2:1:two_one`, or even represent the list of natural numbers as `from 0`, where `from n = n:from(n+1)`. In this way, functions which only need to inspect a finite portion of the structure, e.g., getting the i -th element, can be correctly implemented. On the other hand, functions which need to inspect the whole structure, e.g., `min` getting the minimal element, or `allPos` checking that all elements are positive, have an undefined result (that is, non-termination, operationally).



© Davide Ancona, Pietro Barbieri, Francesco Dagnino, and Elena Zucca;
licensed under Creative Commons License CC-BY

34th European Conference on Object-Oriented Programming (ECOOP 2020).

Editors: Robert Hirschfeld and Tobias Pape; Article No. 1; pp. 1:1–1:28

Leibniz International Proceedings in Informatics



LIPICs Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

More recently, a different, in a sense complementary¹, approach has been considered [17, 8, 3], which focuses on *cyclic* structures (e.g., cyclic lists, trees and graphs). They can be regarded as a particular case of infinite structures: abstractly, they correspond to regular terms (or trees), that is, finitely branching trees whose depth can be infinite, but contain only a finite set of subtrees. For instance, the list `two_one` is regular, whereas the list of natural numbers is not. Typically, cyclic data structures are handled by programming languages by relying on imperative features or ad hoc data structures for bookkeeping. For instance, we can build a cyclic object by assigning to a field of an object a reference to the object itself, or we can visit a graph by marking already encountered nodes. In this approach [17, 8, 3], instead, the programming language natively supports regular structures, as outlined below:

- Data constructors are enriched by allowing equations, e.g., $x = 2 : 1 : x$.
- Functions are *regularly corecursive*, that is, execution keeps track of pending function calls, so that, when the same call is encountered the second time, this is detected, avoiding non-termination as with ordinary recursion. For instance, when calling `min` on the list $x = 2 : 1 : x$, after an intermediate call on the list $y = 1 : 2 : y$, the same call is encountered. Regular corecursion originates from *co-SLD resolution* [20, 21, 7], where already encountered goals (up to unification), called *coinductive hypotheses*, are considered successful. However, co-SLD resolution is not flexible enough to correctly express certain predicates on regular terms; for instance, in the `min` example, the intuitively correct corecursive definition is not sound, because the predicate succeeds for all lower bounds of l , as shown in the following.

When moving from goals to functions calls, the same problem manifests more urgently because a result should always be provided for already encountered calls. To solve this issue, the mechanism of *flexible regular corecursion* can be adopted to allow the programmer to correctly specify the behaviour of recursive functions on cyclic structures. For instance, for function `min`, the programmer specifies that the head of the list should be returned when detecting a cyclic call; in this way, on the list $x = 2 : 1 : x$, the result of the cyclic call is 2, so that the result of the original call is 1, as expected.

Flexible regular corecursion as outlined above has been proposed in the object-oriented [8], functional [17], and logic [3] paradigms (see Section 7 for more details). However, none of these proposals provides formal arguments for the correctness of the given operational semantics, by proving that it is sound with respect to some model of the behaviour of functions (or predicates) on infinite structures. The aim of this paper is to bridge this gap, by providing solid foundations for a programming paradigm natively supporting cyclic data structures. This is achieved thanks to the recently introduced framework of *inference systems with corules* [4, 13], allowing definitions which are neither inductive, nor purely coinductive. We present the approach in the context of Java-like languages, namely on an extension of Featherweight Java (FJ) [15] called coFJ, outlined as follows:

- FJ objects are smoothly generalized from finite to infinite by interpreting their definition coinductively, and methods are equipped with a *codefinition* (an alternative body).
- We provide an abstract big-step semantics for coFJ by an inference system with corules. In coFJ with this semantics, FJ recursive methods on finite objects can be extended to infinite objects as well, and behave as desired by the programmer, by specifying a codefinition. For instance, if the codefinitions for `min` and `allPos` are specified to return the head, and `true`, respectively, then `min` returns 1 on $x = 2 : 1 : x$, and 0 on the list of the natural numbers, whereas `allPos` returns `true` on both lists.

¹ As we will discuss further in the Conclusion.

- Then, we provide an operational (hence, executable) semantics where infinite objects are restricted to regular ones and methods are regularly corecursive, and we show that such operational semantics is sound with respect to the abstract one.

At https://person.dibris.unige.it/zucca-elena/coFJ_implementation.zip we provide a prototype implementation of COFJ, briefly described in the Conclusion. A preliminary version of the operational semantics, with no soundness proof with respect to a formal model, has been given in [10]. An extended version of the paper including proofs can be found at <https://arxiv.org/abs/2005.14085>.

Section 1 is a quick introduction to inference systems with corules. Section 2 describes FJ and informally introduces our approach. In Section 3 we define COFJ and its abstract semantics, in Section 4 the operational semantics, in Section 5 we show some advanced examples, and in Section 6 we prove soundness. Finally, we discuss related work and draw conclusions in Section 7 and Section 8, respectively.

1 Inference systems with corules

First we recall standard notions on inference systems [1, 19]. Assuming a *universe* \mathcal{U} of *judgments*, an *inference system* \mathcal{I} is a set of (*inference*) *rules*, which are pairs $\frac{Pr}{c}$, with $Pr \subseteq \mathcal{U}$ the set of *premises*, and $c \in \mathcal{U}$ the *consequence* (a.k.a. *conclusion*). A rule with an empty set of premises is an *axiom*. A *proof tree* (a.k.a. *derivation*) for a judgment j is a tree whose nodes are (labeled with) judgments, j is the root, and there is a node c with children Pr only if there is a rule $\frac{Pr}{c}$.

The *inductive* and the *coinductive interpretation* of \mathcal{I} , denoted $Ind(\mathcal{I})$ and $CoInd(\mathcal{I})$, are the sets of judgments with, respectively, a finite², and a possibly infinite proof tree. In set-theoretic terms, let $F_{\mathcal{I}} : \wp(\mathcal{U}) \rightarrow \wp(\mathcal{U})$ be defined by $F_{\mathcal{I}}(S) = \{c \mid Pr \subseteq S, \frac{Pr}{c} \in \mathcal{I}\}$, and say that a set S is *closed* if $F_{\mathcal{I}}(S) \subseteq S$, *consistent* if $S \subseteq F_{\mathcal{I}}(S)$. Then, it can be proved that $Ind(\mathcal{I})$ is the smallest closed set, and $CoInd(\mathcal{I})$ is the largest consistent set. We write $\mathcal{I} \vdash j$ when j has a finite derivation in \mathcal{I} , that is, $j \in Ind(\mathcal{I})$.

An *inference system with corules*, or *generalized inference system*, is a pair $(\mathcal{I}, \mathcal{I}^{co})$ where \mathcal{I} and \mathcal{I}^{co} are inference systems, whose elements are called *rules* and *corules*, respectively. Corules can only be used in a special way, as defined below.

For a subset S of the universe, let $\mathcal{I}_{\cap S}$ denote the inference system obtained from \mathcal{I} by keeping only rules with consequence in S . Let $(\mathcal{I}, \mathcal{I}^{co})$ be a generalized inference system. Then, its *interpretation* $Gen(\mathcal{I}, \mathcal{I}^{co})$ is defined by $Gen(\mathcal{I}, \mathcal{I}^{co}) = CoInd(\mathcal{I}_{\cap Ind(\mathcal{I} \cup \mathcal{I}^{co})})$.

In proof-theoretic terms, $Gen(\mathcal{I}, \mathcal{I}^{co})$ is the set of judgments that have a possibly infinite proof tree in \mathcal{I} , where all nodes have a finite proof tree in $\mathcal{I} \cup \mathcal{I}^{co}$, that is, the (standard) inference system consisting of rules and corules. We write $(\mathcal{I}, \mathcal{I}^{co}) \vdash j$ when j is derivable in $(\mathcal{I}, \mathcal{I}^{co})$, that is, $j \in Gen(\mathcal{I}, \mathcal{I}^{co})$. Note that $(\mathcal{I}, \emptyset) \vdash j$ is the same as $\mathcal{I} \vdash j$.

We illustrate these notions by a simple example. As usual, sets of rules are expressed by *meta-rules* with side conditions, and analogously sets of corules are expressed by *meta-corules* with side conditions. (Meta-)corules will be written with thicker lines, to be distinguished from (meta-)rules. The following inference system defines the minimum element of a list, where $[x]$ is the list consisting of only x , and $x : u$ the list with head x and tail u .

$$\frac{}{min([x], x)} \quad \frac{min(u, y)}{min(x : u, z)} z = min(x, y).$$

² Under the common assumption that sets of premises are finite, otherwise we should say well-founded.

The inductive interpretation gives the correct result only on finite lists, since for infinite lists an infinite proof is clearly needed. However, the coinductive one fails to be a function. For instance, for L the infinite list $2 : 1 : 2 : 1 : 2 : 1 : \dots$, any judgment $\text{min}(L, x)$ with $x \leq 1$ can be derived, as shown below.

$$\frac{\dots}{\frac{\text{min}(L, 1)}{\text{min}(1:L, 1)}} \quad \frac{\dots}{\frac{\text{min}(L, 0)}{\text{min}(1:L, 0)}}$$

$$\frac{\text{min}(1:L, 1)}{\text{min}(2:1:L, 1)} \quad \frac{\text{min}(1:L, 0)}{\text{min}(2:1:L, 0)}$$

By adding a corule (in this case a coaxiom), wrong results are “filtered out”:

$$\frac{\dots}{\text{min}(x:\epsilon, x)} \quad \frac{\text{min}(u, y)}{\text{min}(x:u, z)} z = \min(x, y) \quad \frac{\dots}{\text{min}(x:u, x)}$$

Indeed, the judgment $\text{min}(2:1:L, 1)$ has the infinite proof tree shown above, and each node has a finite proof tree in the inference system extended by the corule:

$$\frac{\dots}{\frac{\text{min}(L, 1)}{\text{min}(1:L, 1)}} \quad \frac{\dots}{\text{min}(1:L, 1)}$$

$$\frac{\text{min}(1:L, 1)}{\text{min}(2:1:L, 1)} \quad \frac{\text{min}(1:L, 1)}{\text{min}(2:1:L, 1)}$$

The judgment $\text{min}(2:1:L, 0)$, instead, has the infinite proof tree shown above, but has *no finite proof tree* in the inference system extended by the corule. Indeed, since 0 does not belong to the list, the corule can never be applied. On the other hand, the judgment $\text{min}(L, 2)$ has a finite proof tree with the corule, but cannot be derived since they it has no infinite proof tree. We refer to [4, 5, 6, 13] for other examples.

As final remark, note that requiring the existence of a finite proof tree with corules only for the root is not enough. For regular proof trees, the requirement to have such a proof tree for each node can be simplified in two ways:

- either requiring a sufficiently large finite proof-with-corules for the root, that is, a finite proof tree for the root which includes all the nodes of the regular proof tree
- or requiring a finite proof-with-corules for one node taken from each infinite path.

Let $(\mathcal{I}, \mathcal{I}^{co})$ be a generalized inference system. The *bounded coinduction principle* [4], a generalization of the standard coinduction principle, can be used to prove *completeness* of $(\mathcal{I}, \mathcal{I}^{co})$ w.r.t. a set S (for “specification”) of *valid* judgments.

► **Theorem 1** (Bounded coinduction). *If the following two conditions hold:*

1. $S \subseteq \text{Ind}(\mathcal{I} \cup \mathcal{I}^{co})$, that is, each valid judgment has a finite proof tree in $\mathcal{I} \cup \mathcal{I}^{co}$;
2. $S \subseteq F_{\mathcal{I}}(S)$, that is, each valid judgment is the consequence of a rule in \mathcal{I} with premises in S then $S \subseteq \text{Gen}(\mathcal{I}, \mathcal{I}^{co})$.

2 From FJ to coFJ

We recall FJ, and informally explain its extension with infinite objects and codefinitions.

Featherweight Java. The standard syntax and semantics in big-step style of FJ are shown in Figure 1. We omit cast since this feature does not add significant issues. We adopt a big-step, rather than a small-step style as in the original FJ definition, since in this way the semantics is directly defined by an inference system, denoted \mathcal{I}_{FJ} in the following, which will be equipped with corules to support infinite objects. We write \overline{cd} as metavariable for $cd_1 \dots cd_n$, $n \geq 0$, and analogously for other sequences. We sometimes use the wildcard $_$ when the corresponding metavariable is not relevant.

cd	$::=$	<code>class C extends C' { \overline{fd} \overline{md} }</code>	class declaration
fd	$::=$	<code>C f;</code>	field declaration
md	$::=$	<code>C $m(C_1 x_1, \dots, C_n x_n)$ { e }</code>	method declaration
$e \in \mathcal{E}$	$::=$	<code>x $e.f$ <code>new $C(\overline{e})$ $e.m(\overline{e})$</code></code>	expression
<hr/>			
$v \in \mathcal{V}$	$::=$	<code>new $C(\overline{v})$</code>	(finite) object
<hr/>			
$(FJ\text{-FIELD})$	$\frac{e \Downarrow v}{e.f \Downarrow v_i}$	$v = \text{new } C(v_1, \dots, v_n)$ $fields(C) = f_1 \dots f_n$ $f = f_i, i \in 1..n$	$(FJ\text{-NEW})$ $\frac{\overline{e} \Downarrow \overline{v}}{\text{new } C(\overline{e}) \Downarrow \text{new } C(\overline{v})}$
$(FJ\text{-INVK})$	$\frac{e_0 \Downarrow v_0 \quad \overline{e} \Downarrow \overline{v} \quad e[v_0/\text{this}][\overline{v}/\overline{x}] \Downarrow v}{e_0.m(\overline{e}) \Downarrow v}$	$v_0 = \text{new } C(_)$ $mbody(C, m) = (\overline{x}, e)$	

■ **Figure 1** FJ syntax and big-step rules.

A sequence of class declarations \overline{cd} is called a *class table*. Each class has a canonical constructor whose parameters match the fields of the class, the inherited ones first. We assume standard FJ constraints, e.g., no field hiding and no method overloading. The only variables occurring in method bodies are parameters (including `this`). Values are *objects*, that is, constructor invocations where arguments are values in turn.

The judgment $e \Downarrow v$ is implicitly parameterized on a fixed class table. In the rules we use standard FJ auxiliary functions, omitting their formal definition. Notably, $fields(C)$ returns the sequence $f_1 \dots f_n$ of the field names³ of the class, in declaration order with the inherited first, and $mbody(C, m)$, for method m of the class, the pair of the sequence of parameters and the definition. Substitution $e[\overline{v}/\overline{x}]$, for \overline{e} and \overline{x} of the same length, is defined in the customary manner. Finally, for $\overline{e} = e_1 \dots e_n$ and $\overline{v} = v_1 \dots v_n$, $\overline{e} \Downarrow \overline{v}$ is an abbreviation for $e_1 \Downarrow v_1 \dots e_n \Downarrow v_n$.

Rule (FJ-FIELD) models field access. If the selected field is actually a field of the receiver's class, then the corresponding value is returned as result. Rule (FJ-NEW) models object creation: if the argument expressions \overline{e} evaluate to values \overline{v} , then the result is an object of class C . Rule (FJ-INVK) models method invocation. The receiver and argument expressions are evaluated first. Then, method look-up is performed, starting from the receiver's class, by the auxiliary function $mbody$. Lastly, the definition e of the method, where `this` is replaced by the receiver, and the parameters by the arguments, is evaluated, and its result is returned.

Infinite objects and codefinitions. We take as running example the following FJ implementation of lists of integers, equipped with some typical methods: `isEmpty` tests the emptiness, `incr` returns the list where all elements have been incremented by one, `allPos` checks whether all elements are positive, `member` checks whether the argument is in the list, and `min` returns the minimal element.

³ We omit types since not relevant here. We discuss about type systems for COFJ in the conclusion.

```

class List extends Object {
  bool isEmpty() {true}
  List incr() {new EmptyList()}
  bool allPos() {true}
  bool member(int x) {false}
}
class EmptyList extends List { }
class NonEmptyList extends List {
  int head; List tail;
  bool isEmpty() {false}
  List incr() {new NonEmptyList(this.head+1,this.tail.incr())}
  bool allPos() {if (this.head<=0) false else this.tail.allPos()}
  bool member(int x) {if (this.head==x) true else this.tail.member(x)}
  int min() {
    if (this.tail.isEmpty()) this.head
    else Math.min(this.tail.min(),this.head)
  }
}

```

We used some additional standard constructs, such as conditional and primitive types `bool` and `int` with their operations; to avoid to use abstract methods, `List` provides the default implementation on empty lists, overridden in `NonEmptyList`, except for method `min` which is only defined on non empty lists.

In FJ we can represent finite lists. For instance, the object

```
new NonEmptyList(2, new NonEmptyList(1, new EmptyList()))
```

which we will abbreviate $[2, 1]$, represents a list of two elements, and it is easy to see that all the above method definitions provide the expected meaning on finite lists.

On the other hand, since the syntactic definition for objects is interpreted, like the others, inductively, in FJ objects are *finite*, hence we cannot represent, e.g., the infinite list of natural numbers $[0, 1, 2, 3, \dots]$, abbreviated $[0..]$, or the infinite list $[2, 1, 2, 1, 2, 1, \dots]$, abbreviated $[2, 1]^\omega$. To move from finite to infinite objects, it is enough to interpret the syntactic definition for values *coinductively*, so to obtain infinite terms as well. However, to make the extension significant, we should be able to *generate* such infinite objects as results of expressions, and to appropriately *handle* them by methods.

To generate infinite objects, e.g., the infinite lists mentioned above, a natural approach is to consider method definitions as *corecursive*, that is, to take the *coinductive* interpretation of the inference system in Figure 1. Consider the following class:

```

class ListFactory extends Object {
  NonEmptyList from(int x) {new NonEmptyList(x, this.from(x+1))}
  NonEmptyList two_one() {new NonEmptyList(2, this.one_two())}
  NonEmptyList one_two() {new NonEmptyList(1, this.two_one())}
}

```

With the standard FJ semantics, given by the inductive interpretation of the inference system in Figure 1, the method invocation `new ListFactory().from(0)` (abbreviated `from0` in the following) has no result, since there is no finite proof tree for a judgment of shape `from0 ↓`. Taking the coinductive interpretation, instead, such call returns as result the infinite list of natural numbers $[0..]$, since there is an infinite proof tree for the judgment `from0 ↓ [0..]`. Analogously, the method invocation `new ListFactory().two_one()` returns $[2, 1]^\omega$. Moreover, the method invocations `[0..].incr()` and `[2, 1]^\omega.incr()` correctly return as result the infinite lists $[1..]$ and $[3, 2]^\omega$, respectively.

However, in many cases to consider method definitions as corecursive is not satisfactory, since it leads to non-determinism, as shown for inference systems in Section 1. For instance, for the method invocation `[0..].allPos()` both judgments $[0..].\text{allPos}() \Downarrow \text{true}$ and $[0..].\text{allPos}() \Downarrow \text{false}$ are derivable, and analogously for $[2,1]^\omega.\text{allPos}()$. In general, both results can be obtained for any infinite list of all positive numbers. A similar behavior is exhibited by method `member`: given an infinite list L which does not contain x , both judgments $L.\text{member}(x) \Downarrow \text{true}$ and $L.\text{member}(x) \Downarrow \text{false}$ are derivable. Finally, for the method invocation $[2,1]^\omega.\text{min}()$, any judgment $[2,1]^\omega.\text{min}() \Downarrow x$ with $x \leq 1$ can be derived.

To solve this problem, COFJ allows the programmer to *control* the semantics of corecursive methods by adding a *codefinition*⁴, that is, an alternative method body playing a special role. Depending on the codefinition, the purely coinductive interpretation is refined, by filtering out some judgments. In the example, to achieve the expected meaning, the programmer should provide the following codefinitions.

```
class ListFactory extends Object {
  NonEmptyList from(int x) {
    new NonEmptyList(x, this.from(x+1)) corec {any}
  }
  NonEmptyList one_two() {
    new NonEmptyList(1, this.two_one()) corec {any}
  }
  NonEmptyList two_one() {
    new NonEmptyList(2, this.one_two()) corec {any}
  }
}
class NonEmptyList extends List {
  int head; List tail;
  bool isEmpty() {false}
  List incr() {
    new NonEmptyList(this.head+1, this.tail.incr()) corec {any}
  }
  bool allPos() {
    if (this.head <= 0) false else this.tail.allPos() corec {true}
  }
  bool member(int x) {
    if (this.head == x) true else this.tail.member(x) corec {false}
  }
  int min() {
    if (this.tail.isEmpty()) this.head
    else Math.min(this.tail.min(), this.head)
  } corec {this.head}
}
```

For the three methods of `ListFactory` and for the method `incr` the codefinition is `any`. This corresponds to keeping the coinductive interpretation as it is, as is appropriate in these cases since it provides only the expected result. In the other three methods, instead, the effect of the codefinition is to filter the results obtained by the coinductive interpretation. The way this is achieved is explained in the following section. Finally, for method `isEmpty` no codefinition is added, since the inductive behaviour works on infinite lists as well.

3 coFJ and its abstract semantics

We formally define COFJ, illustrate how the previous examples get the expected semantics, and show that, despite its non-determinism, COFJ is a conservative extension of FJ.

⁴ The term “codefinition” is meant to suggest “alternative definition used to handle corecursion”.

1:8 Sound Regular Corecursion in coFJ

cd	$:: =$	<code>class C extends C' { \overline{fd} \overline{md} }</code>	class declaration
fd	$:: =$	<code>C f;</code>	field declaration
md	$:: =$	<code>C $m(C_1 x_1, \dots, C_n x_n)$ { e } [corec { e' }]</code>	method declaration with codefinition
$e \in \mathcal{E}$	$:: =$	<code>x $e.f$ new $C(\overline{e})$ $e.m(\overline{e})$</code>	expression
$v \in \mathcal{V}^a$	$:: =_{\text{co}}$	<code>new $C(\overline{v})$</code>	possibly infinite object
$e \in \mathcal{E}^a$	$:: =$	<code>x $e.f$ new $C(\overline{e})$ $e.m(\overline{e})$ v</code>	runtime expression

$$\begin{array}{c}
\text{(ABS-FIELD)} \frac{e \Downarrow v}{e.f \Downarrow v_i} \quad \begin{array}{l} v = \text{new } C(v_1, \dots, v_n) \\ \text{fields}(C) = f_1 \dots f_n \\ f = f_i, i \in 1..n \end{array} \quad \text{(ABS-NEW)} \frac{\overline{e} \Downarrow \overline{v}}{\text{new } C(\overline{e}) \Downarrow \text{new } C(\overline{v})} \\
\text{(ABS-INVK)} \frac{e_0 \Downarrow v_0 \quad \overline{e} \Downarrow \overline{v} \quad e[v_0/\text{this}][\overline{v}/\overline{x}] \Downarrow v}{e_0.m(\overline{e}) \Downarrow v} \quad \begin{array}{l} v_0 = \text{new } C(_) \\ \text{mbody}(C, m) = (\overline{x}, e) \end{array} \quad \text{(ABS-CO-VAL)} \frac{}{v \Downarrow v} \\
\text{(ABS-CO-INVK)} \frac{e_0 \Downarrow v_0 \quad \overline{e} \Downarrow \overline{v} \quad e'[v_0/\text{this}][\overline{v}/\overline{x}][v/\text{any}] \Downarrow v_{co}}{e_0.m(\overline{e}) \Downarrow v_{co}} \quad \begin{array}{l} v_0 = \text{new } C(_) \\ \text{co-mbody}(C, m) = (\overline{x}, e') \end{array}
\end{array}$$

■ **Figure 2** coFJ syntax and abstract semantics.

Formal definition of coFJ. The coFJ syntax is given in Figure 2. As the reader can note, the only difference is that method declarations include now, besides a definition e , an optional *codefinition* e' , as denoted by the square brackets in the production. Furthermore, besides **this**, there is another special variable **any**, which can only occur in codefinitions. The codefinition will be used to provide an abstract semantics through an inference system with corules, where the role of **any** is to be a placeholder for an arbitrary value. For simplicity, we require the codefinition e' to be statically restricted to avoid recursive (even indirect) calls to the same method (we omit the standard formalization). Note that FJ is a (proper) subset of coFJ: indeed, an FJ class table is a coFJ class table with no codefinitions.

The syntactic definition for values is the same as before, but is now interpreted *coinductively*, as indicated by the symbol $:: =_{\text{co}}$. In this way, infinite objects are supported. By replacing method parameters by arguments, we obtain *runtime expressions* admitting infinite objects as subterms. The sets \mathcal{V} and \mathcal{E} of FJ objects and expressions are subsets of \mathcal{V}^a and \mathcal{E}^a , respectively. The judgment $e \Downarrow v$, with $e \in \mathcal{E}^a$ and $v \in \mathcal{V}^a$, is defined by an inference system with corules $(\mathcal{I}_{\text{FJ}}, \mathcal{I}_{\text{FJ}}^{\text{co}})$ where the rules \mathcal{I}_{FJ} are those⁵ of FJ, as in Figure 1, and the corules $\mathcal{I}_{\text{FJ}}^{\text{co}}$ are instances of two metacorules.

Corule (ABS-CO-VAL) is needed to obtain a value for infinite objects, as shown below. Corule (ABS-CO-INVK) is analogous to the standard rule for method invocation, but uses the codefinition, and the variable **any** can be non-deterministically substituted with an arbitrary value. The auxiliary function *co-mbody* is defined analogously to *mbody*, but it returns the codefinition. Note that, even when $\text{mbody}(C, m)$ is defined, $\text{co-mbody}(C, m)$ can be undefined since no codefinition has been specified. This can be done to force a purely inductive behaviour for the method.

⁵ To be precise, meta-rules are the same, with meta-variables e and v ranging on \mathcal{E}^a , and \mathcal{V}^a , respectively. However, we could have taken this larger universe in FJ as well without affecting the defined relation.

$$\begin{array}{c}
\frac{\frac{\frac{(\text{ABS-NEW})}{\text{new LF}() \Downarrow \text{new LF}()} \quad (\text{N-VAL})}{n \Downarrow n} \quad (\text{ABS-NEW})}{\text{new NEL}(n, \text{new LF}().\text{from}(n+1)) \Downarrow [n..]} \quad \frac{(\text{N-VAL})}{n \Downarrow n} \quad T_n}{\text{from}_n \Downarrow [n..]} \\
T_n = (\text{ABS-INVK}) \\
\\
\frac{\frac{\frac{(\text{ABS-NEW})}{\text{new LF}() \Downarrow \text{new LF}()} \quad (\text{N-VAL})}{n+1 \Downarrow n+1} \quad \dots \quad (\text{ABS-NEW})}{\text{new NEL}(n+1, \text{new LF}().\text{from}(n+1+1)) \Downarrow [n+1..]} \quad \frac{(\text{N-VAL})}{n+1 \Downarrow n+1} \quad T_{n+2}}{\text{new LF}().\text{from}(n+1) \Downarrow [n+1..]} \\
T_{n+1} = (\text{ABS-INVK}) \\
\\
\frac{\frac{(\text{ABS-NEW})}{\text{new LF}() \Downarrow \text{new LF}()} \quad (\text{N-VAL})}{n \Downarrow n} \quad (\text{ABS-CO-VAL})}{[n..] \equiv \text{any}[\text{new LF}()/\text{this}][[n..]/\text{any}] \Downarrow [n..]} \\
(\text{ABS-CO-INVK}) \\
\text{from}_n \Downarrow [n..]
\end{array}$$

■ **Figure 3** Infinite (top) and finite (bottom) proof trees for $\text{from}_n \Downarrow [n..]$.

Examples. As an example, we illustrate in Figure 3 the role of the two corules for the call `new ListFactory().from(0)`. For brevity, we write abbreviated class names. Furthermore, from_n stands for the call `new ListFactory().from(n)` and $[n..]$ for the infinite object `new NonEmptyList(n, new NonEmptyList(n+1, ...))`.

In the top part of Figure 3, we show the infinite proof tree T_n which can be constructed, for any natural number n , for the judgment $\text{from}_n \Downarrow [n..]$ without the use of corules. We use standard rules (N-VAL) and (+) to deal with integer constants and addition.

To derive the judgment in the inference system with corules, each node in this infinite tree should have a finite proof tree with the corules. Notably, this should hold for nodes of shape $\text{from}_n \Downarrow [n..]$, and indeed the finite proof tree for such nodes is shown in the bottom part of the figure. Note that, in this example, the result for the call from_n is uniquely determined by the rules, hence the role of the corules is just to “validate” this result. To this end, the codefinition of the method `from` is the special variable `any`, which, when evaluating the codefinition, can be replaced by any value, hence, in particular, by the correct result $[n..]$. Corule (ABS-CO-VAL) is needed to obtain a finite proof tree for the infinite objects of shape $[n..]$. Analogous infinite and finite proof trees can be constructed for the judgments `new ListFactory().two_one() \Downarrow [2, 1]^\omega`, `[0..].incr() \Downarrow [1..]` and `[2, 1]^\omega.incr() \Downarrow [3, 2]^\omega`.

For the method call `[0..].allPos()`, instead, both judgments `[0..].allPos() \Downarrow true` and `[0..].allPos() \Downarrow false` have an infinite proof tree. However, no finite proof tree using the codefinition can be constructed for the latter, whereas this is trivially possible for the former. Analogously, given an infinite list L which does not contain x , only the judgment `L.member(x) \Downarrow false` has a finite proof tree using the codefinition.

Finally, for the method invocation `[2, 1]^\omega.min()`, for any $v \leq 1$ there is an infinite proof tree built without corules for the judgment `[2, 1]^\omega.min() \Downarrow v` as shown in Figure 4. However, only the judgment `[2, 1]^\omega.min() \Downarrow 1` has a finite proof tree using the codefinition (Figure 5). For space reasons in both figures ellipses are used to omit the less interesting parts of the proof trees; we use the standard rule (IF-F) for conditional, and the predefined function `Math.min` on integers.

Non-determinism and conservativity. The COFJ abstract semantics is inherently non-deterministic. Indeed, depending on the codefinition, the non-determinism of the coinductive interpretation may be kept. For instance, consider the following method declaration:

$$\begin{array}{c}
 \frac{(ABS-INVK) \frac{T_0 \quad T_1}{[2, 1]^\omega . \text{min}() \Downarrow v}}{T_0 = (ABS-NEW) \frac{(N-VAL) \frac{2 \Downarrow 2}{\overline{2 \Downarrow 2}} \quad (ABS-NEW) \frac{(N-VAL) \frac{1 \Downarrow 1}{\overline{1 \Downarrow 1}} \quad T_0}{[1, 2]^\omega \Downarrow [1, 2]^\omega}}{[2, 1]^\omega \Downarrow [2, 1]^\omega}} \\
 \\
 T_1 = (IF-F) \frac{\frac{[2, 1]^\omega . \text{tail.isEmpty}() \Downarrow \text{false} \quad \frac{T_2 \quad \vdots}{[2, 1]^\omega . \text{tail.min}() \Downarrow v} \quad \frac{\vdots \quad \vdots}{[2, 1]^\omega . \text{head} \Downarrow 2}}{\text{Math.min}([2, 1]^\omega . \text{tail.min}(), [2, 1]^\omega . \text{head}) \Downarrow v}}{\text{if } [2, 1]^\omega . \text{tail.isEmpty}() \text{ then } [2, 1]^\omega . \text{head} \text{ else } \text{Math.min}([2, 1]^\omega . \text{tail.min}(), [2, 1]^\omega . \text{head}) \Downarrow v}} \\
 \\
 T_2 = (IF-F) \frac{\frac{[1, 2]^\omega . \text{tail.isEmpty}() \Downarrow \text{false} \quad \frac{T_1 \quad \vdots}{[1, 2]^\omega . \text{tail.min}() \Downarrow v} \quad \frac{\vdots \quad \vdots}{[1, 2]^\omega . \text{head} \Downarrow 1}}{\text{Math.min}([1, 2]^\omega . \text{tail.min}(), [1, 2]^\omega . \text{head}) \Downarrow v}}{\text{if } [1, 2]^\omega . \text{tail.isEmpty}() \text{ then } [1, 2]^\omega . \text{head} \text{ else } \text{Math.min}([1, 2]^\omega . \text{tail.min}(), [1, 2]^\omega . \text{head}) \Downarrow v}}
 \end{array}$$

■ **Figure 4** Infinite proof tree for $[2, 1]^\omega . \text{min}() \Downarrow v$ with $v \leq 1$ (main tree at the top left corner).

$$\begin{array}{c}
 \frac{(ABS-INVK) \frac{T_0 \quad (IF-F) \frac{\frac{[2, 1]^\omega . \text{tail.isEmpty}() \Downarrow \text{false} \quad \frac{\frac{\frac{\dots}{[1, 2]^\omega \Downarrow [1, 2]^\omega} \quad (ABS-CO-VAL) \frac{[1, 2]^\omega \Downarrow [1, 2]^\omega}{[1, 2]^\omega . \text{head} \Downarrow 1}}{\vdots}}{[2, 1]^\omega . \text{tail.min}() \Downarrow 1}}{\text{Math.min}([2, 1]^\omega . \text{tail.min}(), [2, 1]^\omega . \text{head}) \Downarrow 1}}{\text{if } [2, 1]^\omega . \text{tail.isEmpty}() \text{ then } [2, 1]^\omega . \text{head} \text{ else } \text{Math.min}([2, 1]^\omega . \text{tail.min}(), [2, 1]^\omega . \text{head}) \Downarrow 1}}{[2, 1]^\omega . \text{min}() \Downarrow 1}}{[2, 1]^\omega . \text{min}() \Downarrow 1}}
 \end{array}$$

■ **Figure 5** Finite proof tree with codefinition for $[2, 1]^\omega . \text{min}() \Downarrow 1$ (T_0 as in Figure 4).

```

class C {
  C m() { this.m() } corec { any }
}
    
```

Method $m()$ recursively calls itself. In the abstract semantics, the judgment $\text{new } C().m() \Downarrow v$ can be derived for any value v . In the operational semantics defined in Section 4, such method call evaluates to $(x, x : x)$, that is, the representation of *undetermined*.

However, determinism of FJ evaluation is preserved. Indeed, COFJ abstract semantics is a *conservative* extension of FJ semantics, as formally stated below.

► **Theorem 2 (Conservativity).** *If $\mathcal{I}_{FJ} \vdash e \Downarrow v$, then $(\mathcal{I}_{FJ}, \mathcal{I}_{FJ}^{co}) \vdash e \Downarrow v'$ iff $v = v'$.*

Proof. Both directions can be easily proved by induction on the definition of $\mathcal{I}_{FJ} \vdash e \Downarrow v$. For the left-to-right direction, the fact that each syntactic category has a unique applicable meta-rule is crucial. ◀

This theorem states that, whichever the codefinitions chosen, COFJ does not change the semantics of expressions evaluating to some value in FJ. That is, COFJ abstract semantics allows derivation of new values only for expressions whose semantics is undefined in standard FJ, as in the examples shown above. Note also that, if no codefinition is specified, then the COFJ abstract semantics *coincides* with the FJ one, because corule (ABS-CO-INVK) cannot be applied, hence no infinite proof trees can be built for the evaluation of FJ expressions.

4 Operational semantics

We informally introduce the operational semantics of COFJ, provide its formal definition, and prove that it is deterministic and conservative.

Outline. In contrast to the abstract semantics of the previous section, the aim is to define a semantics which leads to an interpreter for the calculus. To obtain this, there are two issues to be considered:

1. infinite (regular) objects should be represented in a finite way;
2. infinite (regular) proof trees should be replaced by finite proof trees.

In the following we explain how these issues are handled in the COFJ operational semantics.

To obtain (1), we use an approach based on *capsules* [16], which are essentially expressions supporting cyclic references. In our context, capsules are pairs (e, σ) where e is an FJ expression and σ is an *environment*, that is, a finite mapping from variables into FJ expressions. Moreover, the following *capsule property* is satisfied: writing $FV(e)$ for the set of free variables in e , $FV(e) \subseteq \text{dom}(\sigma)$ and, for all $x \in \text{dom}(\sigma)$, $FV(\sigma(x)) \subseteq \text{dom}(\sigma)$. An FJ source expression e is represented by the capsule (e, \emptyset) , where \emptyset denotes the empty environment. In particular, values are pairs (v, σ) where v is an *open* FJ object, that is, an object possibly containing variables. In this way, cyclic objects can be obtained: for instance, $(x, x : \text{new NEL}(2, \text{new NEL}(1, x)))$ represents the infinite regular list $[2, 1]^\omega$ considered before.

To obtain (2), methods are *regularly corecursive*. This means that execution keeps track of the pending method calls, so that, when a call is encountered the second time, this is detected⁶, avoiding non-termination as it would happen with ordinary recursion. Regular corecursion in COFJ is *flexible*, since the behaviour of the method when a cycle is detected is specified by the codefinition.

Consider, for instance, the method call `new ListFactory().two_one()`; thanks to regular corecursion, the result is the cyclic object $(x, x : \text{new NEL}(2, \text{new NEL}(1, x)))$. Indeed, the operational semantics associates a fresh variable, say, x , to the initial call, so that, when the same call is encountered the second time, the association $x : x$ is added in the environment, and the codefinition is evaluated where `any` is replaced by x . Hence, $(x, x : x)$ is returned as result, so that the result of the original call is $(x, x : \text{new NEL}(2, \text{new NEL}(1, x)))$. The call `new ListFactory().from(0)`, instead, does not terminate in the operational semantics, since no call is encountered more than once (the resulting infinite object is non-regular).

Consider now the call $[2, 1]^\omega.\text{allPos}()$. In this case, when the call is encountered the second time, after an intermediate call $[1, 2]^\omega.\text{allPos}()$, the result of the evaluation of the codefinition is `true`, so that the result of the original call is `true` as well.⁷ If the codefinition were `any`, then the result would be $(x, x : x)$, that is, undetermined. Note that, if the list is finite, then no regular corecursion is involved, since the same call cannot occur more than once; the same holds if the list is cyclic, but contains a non-positive element, hence the method invocation returns `false`. The only case requiring regular corecursion is when the method is invoked on a cyclic list with all positive elements, as $[2, 1]^\omega$.

In the case of $[2, 1]^\omega.\text{min}()$, when the call is encountered the second time the result of the evaluation of the codefinition is 2, so that the result of the intermediate call $[1, 2]^\omega.\text{min}()$ is 1, and this is also the result of the original call.

⁶ The semantics detects an already encountered call by relying on capsule equivalence (Figure 7).

⁷ To be rigorous, a capsule of shape $(\text{true}, _)$.

Formal definition. To formally express the approach described above, the judgment of the operational semantics has shape $e, \sigma, \tau \Downarrow v, \sigma'$ where: (e, σ) is the capsule to be evaluated; τ is a *call trace*, used to keep track of already encountered calls, that is, an injective map from calls $v_0.m(\bar{v})$ to (possibly tagged) variables, and (v, σ') is the capsule result. Variables in the codomain of the call trace have a tag ck during the checking step for the corresponding call, as detailed below. The pair (e, σ) and (v, σ') are assumed to satisfy the capsule property.

The semantic rules are given in Figure 6. We denote by $\sigma\{x:v\}$ the environment which gives v on x , and is equal to σ elsewhere, and analogously for other maps. Furthermore, we use the following notations, formally defined in Figure 7.

- $unfold(v, \sigma)$ is the *unfolding* of v in σ , that is, the corresponding object, if any.
- $\sigma_1 \sqcup \sigma_2$ is the *union of environments*, defined if they agree on the common domain.
- $(v, \sigma) \approx (v', \sigma')$ is the *equivalence of capsules*. As will be formalized in the first part of Section 6, equivalent capsules denote the same sets of abstract objects. This equivalence is extended by congruence to expressions, in particular to calls $v_0.m(\bar{v})$.
- $\tau_{\approx\sigma}$ is obtained by extending τ *up to equivalence* in σ . That is, detection of already encountered calls is performed up-to equivalence in the current environment.

Rule (VAL) is needed for objects which are not FJ objects. Rule (FIELD) is similar to that of FJ except that the capsule (v, σ') must be unfolded to retrieve the corresponding object. Furthermore, the resulting environment is that obtained by evaluating the receiver. Rule (NEW) is analogous to that of FJ. The resulting environment is the union of those obtained by evaluating the arguments.

There are four rules for method invocation. In all of them, as in the FJ rule, the receiver and argument expressions are evaluated first to obtain the call $c = v.m(\bar{v})$. The environment $\hat{\sigma}$ is the union of those obtained by these evaluations. Then, the behavior is different depending whether such call (meaning a call equivalent to c in $\hat{\sigma}$) has been already encountered.

Rules (INVK-OK) and (INVK-CHECK) handle⁸ a call c which is encountered the first time, as expressed by the side condition $c \notin dom(\tau_{\approx\hat{\sigma}})$. In both, the definition e , where the receiver replaces **this** and the arguments replace the parameters, is evaluated. Such evaluation is performed in the call trace τ updated to associate the call c with an unused variable x (in these two rules “ x fresh” means that x does not occur in the derivations of $e_i, \sigma, \tau \Downarrow v_i, \sigma'_i$, for all $i \in 0..n$), and produces the capsule (v, σ') . Then there are two cases, depending on whether $x \in dom(\sigma')$ holds.

If $x \notin dom(\sigma')$, then the evaluation of the definition for c has been performed without evaluating the codefinition. That is, the same call has not been encountered, hence the result has been obtained by standard recursion, and no additional check is needed.

If $x \in dom(\sigma')$, instead, then the evaluation of the definition for c has required to evaluate the codefinition. In this case, an additional check is required (third premise). That is, $e[v_0/\mathbf{this}][\bar{v}/\bar{x}]$ is evaluated once more under the assumption that v is the result of the call. Formally, evaluation takes place in an environment updated to associate x with v , and the variable x corresponding to the call is tagged with ck . The capsule result obtained in this way must be (equivalent to) that obtained by the first evaluation of the body of the method. In Section 5 we discuss in detail the role of this additional check, showing an example where it is necessary. If the check succeeds, then the final result is the variable x in the environment updated to associate x with v . Otherwise, rule (INVK-CHECK) cannot be applied since the last premise does not hold. For simplicity, we assume the result of c to be undefined in this case; an additional rule could be added raising a runtime error in case the result is different from the expected one, as should be done in an implementation.

⁸ The two rules could be merged together, but we prefer to make explicit the difference for sake of clarity.

$v \in \mathcal{V}^{\text{OP}}$::=	$\text{new } C(\bar{v}) \mid x$	open object
σ	::=	$x_1 : v_1 \dots x_n : v_n \quad (n \geq 0)$	environment
c	::=	$v.m(\bar{v})$	call
t	::=	$[\text{ck}]$	optional checking tag
τ	::=	$c_1 : x_1^{t_1}, \dots, c_n : x_n^{t_n} \quad (n \geq 0)$	call trace

$$\frac{}{(\text{VAL}) \quad v, \sigma, \tau \Downarrow v, \sigma} \quad \frac{e, \sigma, \tau \Downarrow v, \sigma' \quad \text{unfold}(v, \sigma') = \text{new } C(v_1, \dots, v_n)}{(\text{FIELD}) \quad e.f, \sigma, \tau \Downarrow v_i, \sigma'} \quad \begin{array}{l} \text{fields}(C) = f_1 \dots f_n \\ f = f_i, i \in 1..n \end{array}$$

$$\frac{e_i, \sigma, \tau \Downarrow v_i, \sigma'_i \quad \forall i \in 1..n}{(\text{NEW}) \quad \text{new } C(e_1, \dots, e_n), \sigma, \tau \Downarrow \text{new } C(v_1, \dots, v_n), \sqcup_{i \in 1..n} \sigma'_i}$$

In all the following rules:

$$\begin{array}{l} \bar{e} = e_1, \dots, e_n \\ \bar{v} = v_1 \dots v_n \\ c = v_0.m(\bar{v}) \\ \hat{\sigma} = \sqcup_{i \in 0..n} \sigma'_i \\ \text{unfold}(v_0, \sigma'_0) = \text{new } C(_) \end{array}$$

$$\frac{e_i, \sigma, \tau \Downarrow v_i, \sigma'_i \quad \forall i \in 0..n \quad e[v_0/\text{this}][\bar{v}/\bar{x}], \hat{\sigma}, \tau\{c:x\} \Downarrow v, \sigma'}{(\text{INVK-OK}) \quad e_0.m(\bar{e}), \sigma, \tau \Downarrow v, \sigma'} \quad \begin{array}{l} c \notin \text{dom}(\tau_{\approx \hat{\sigma}}) \\ x \text{ fresh} \\ \text{mbody}(C, m) = (\bar{x}, e) \\ x \notin \text{dom}(\sigma') \end{array}$$

$$\frac{e_i, \sigma, \tau \Downarrow v_i, \sigma'_i \quad \forall i \in 0..n \quad e[v_0/\text{this}][\bar{v}/\bar{x}], \hat{\sigma}, \tau\{c:x\} \Downarrow v, \sigma' \quad e[v_0/\text{this}][\bar{v}/\bar{x}], \hat{\sigma} \sqcup \sigma'\{x:v\}, \tau\{c:x^{\text{ck}}\} \Downarrow v', \sigma''}{(\text{INVK-CHECK}) \quad e_0.m(\bar{e}), \sigma, \tau \Downarrow x, \sigma'\{x:v\}} \quad \begin{array}{l} c \notin \text{dom}(\tau_{\approx \hat{\sigma}}) \\ x \text{ fresh} \\ \text{mbody}(C, m) = (\bar{x}, e) \\ x \in \text{dom}(\sigma') \\ (x, \sigma'\{x:v\}) \approx (v', \sigma'') \end{array}$$

$$\frac{e_i, \sigma, \tau \Downarrow v_i, \sigma'_i \quad \forall i \in 0..n \quad e'[v_0/\text{this}][\bar{v}/\bar{x}][x/\text{any}], \hat{\sigma}\{x:x\}, \tau \Downarrow v, \sigma'}{(\text{COREC}) \quad e_0.m(\bar{e}), \sigma, \tau \Downarrow v, \sigma'\{x:x\}} \quad \begin{array}{l} \tau_{\approx \hat{\sigma}}(c) = x \\ \text{co-mbody}(C, m) = (\bar{x}, e') \end{array}$$

$$\frac{e_i, \sigma, \tau \Downarrow v_i, \sigma'_i \quad \forall i \in 0..n}{(\text{LOOK-UP}) \quad e_0.m(\bar{e}), \sigma, \tau \Downarrow x, \hat{\sigma}} \quad \tau_{\approx \hat{\sigma}}(c) = x^{\text{ck}}$$

■ **Figure 6** COFJ operational semantics.

$$\begin{aligned} \text{unfold}(v, \sigma) &= \begin{cases} \mathbf{new} C(\bar{v}) & \text{if } v = \mathbf{new} C(\bar{v}) \\ \text{unfold}(\sigma(v), \sigma) & \text{if } v = x \end{cases} \\ \text{undet}(\sigma) &= \{x \in \text{dom}(\sigma) \mid \text{unfold}(x, \sigma) \uparrow\} \end{aligned}$$

For σ_1 and σ_2 such that $\sigma_1(x) = \sigma_2(x)$ for all $x \in \text{dom}(\sigma_1) \cap \text{dom}(\sigma_2)$

$$(\sigma_1 \sqcup \sigma_2)(x) = \begin{cases} \sigma_1(x) & x \in \text{dom}(\sigma_1) \\ \sigma_2(x) & x \in \text{dom}(\sigma_2) \end{cases}$$

Set $\overset{\sigma}{\leftrightarrow}$ the least equivalence relation on $\text{undet}(\sigma)$ such that $x \overset{\sigma}{\leftrightarrow} y$ if $\sigma(x) = y$, $[x]$ the equivalence class of x , and $\text{undet}_{\leftrightarrow}(\sigma)$ the quotient. A relation $\alpha \subseteq \text{undet}(\sigma_1) \times \text{undet}(\sigma_2)$ is a σ_1, σ_2 -renaming if it induces a (partial) bijection from $\text{undet}_{\leftrightarrow}(\sigma_1)$, still denoted α , to $\text{undet}_{\leftrightarrow}(\sigma_2)$. Given α a σ_1, σ_2 -renaming, the relation $(x, \sigma_1) \approx_{\alpha} (x', \sigma_2)$ is coinductively defined by:

$$\frac{}{(x, \sigma) \approx_{\alpha} (x', \sigma')} \quad x \alpha x' \quad \frac{(\mathbf{v}_i, \sigma) \approx_{\alpha} (\mathbf{v}'_i, \sigma') \quad \forall i \in 1..n \quad \text{unfold}(v, \sigma) = \mathbf{new} C(\mathbf{v}_1, \dots, \mathbf{v}_n)}{(\mathbf{v}, \sigma) \approx_{\alpha} (\mathbf{v}', \sigma')} \quad \text{unfold}(v', \sigma') = \mathbf{new} C(\mathbf{v}'_1, \dots, \mathbf{v}'_n)$$

A σ_1, σ_2 -renaming α is *strict* if, for $x, y \in \text{undet}(\sigma_1) \cap \text{undet}(\sigma_2)$, $[x] \alpha [y]$ iff $x \overset{\sigma_1}{\leftrightarrow} y$ and $x \overset{\sigma_2}{\leftrightarrow} y$. We write $(\mathbf{v}, \sigma) \approx (\mathbf{v}', \sigma')$ if $(\mathbf{v}, \sigma) \approx_{\alpha} (\mathbf{v}', \sigma')$ for some strict α .

$$\tau_{\approx \sigma}(c') = \tau(c) \text{ for each } c' \text{ such that } (c', \sigma) \approx (c, \sigma)$$

■ **Figure 7** coFJ auxiliary definitions.

The remaining rules handle an already encountered call c , that is, $\tau_{\approx \hat{\sigma}}(c)$ is defined. The behaviour is different depending on whether the corresponding variable x is tagged or not.

If x is not tagged, then rule (COREC) evaluates the codefinition where the receiver object replaces **this**, the arguments replace the parameters, and, furthermore, the variable x found in the call trace replaces **any**. In addition, $\hat{\sigma}$ is updated to associate x with x . In this way, the semantics keeps track of the application of rule (COREC).

If x is tagged, instead, then we are in a checking step for the corresponding call. In this case, rule (LOOK-UP) simply returns the associated variable for a call; by definition of the operational semantics, in this case such a variable is always defined in the environment.

Figure 7 contains the formal definitions of the notations used in the rules.

Note that *unfold*, being inductively defined, can be undefined, denoted \uparrow , in presence of unguarded cycles among variables. Capsule equivalence, instead, is defined coinductively, so that, e.g., $(x, x : \mathbf{new} C(x))$ is equivalent to $(x, x : \mathbf{new} C(\mathbf{new} C(x)))$. Capsule equivalence implicitly subsumes α -equivalence of variables whose unfolding is defined, e.g., $(x, x : \mathbf{new} C(x))$ is equivalent to $(y, y : \mathbf{new} C(y))$. Instead, α -equivalence of undetermined variables is given by an explicit renaming, which should preserve disjointness of cycles. For instance, $(\mathbf{new} C(x, y), (x : y, y : x))$ is equivalent to $(\mathbf{new} C(x, x), x : x)$, but is *not* equivalent to $(\mathbf{new} C(x, y), (x : x, y : y))$. Indeed, in the latter case x and y can be instantiated independently. We will prove in Section 6 (Theorem 10) that the relation \approx_{α} , for some σ_1, σ_2 -renaming α , is the operational counterpart of the fact that two capsules denote the same set of abstract values. The stronger strictness condition prevents erroneous identification of objects during evaluation, e.g., $(\mathbf{new} C(x, y), (x : x, y : y))$ is not equivalent to $(\mathbf{new} C(y, x), (y : y, x : x))$.

Determinism and conservativity. In contrast to coFJ abstract semantics, but like FJ, coFJ operational semantics is deterministic.

► **Theorem 3** (Determinism). *If $e, \sigma, \tau_1 \Downarrow v_1, \sigma_1$ and $e, \sigma, \tau_2 \Downarrow v_2, \sigma_2$ hold and $\text{dom}(\tau_1) = \text{dom}(\tau_2)$, then (v_1, σ_1) and (v_2, σ_2) are equal up-to α -equivalence.*

Proof. The proof is by induction on the derivation for $e, \sigma, \tau_1 \Downarrow v_1, \sigma_1$. The key point is that, once fixed e, σ and $\text{dom}(\tau_1)$, there is a unique applicable rule, hence both $e, \sigma, \tau_1 \Downarrow v_1, \sigma_1$ and $e, \sigma, \tau_2 \Downarrow v_2, \sigma_2$ are derived by the same rule. ◀

As the abstract one, the operational semantics is a conservative extension of the standard FJ semantics. This result follows from soundness with respect to the abstract semantics in next section, however the direct proof below provides some useful insight.

► **Theorem 4** (Conservativity). *If $\mathcal{I}_{\text{FJ}} \vdash e \Downarrow v$, then $e, \emptyset, \emptyset \Downarrow v, \sigma$ holds iff $v = v$ and $\sigma = \emptyset$.*

For the proof, we need some auxiliary lemmas and definitions. First, we note that FJ has the *strong determinism* property: each expression has at most one finite proof tree in \mathcal{I}_{FJ} .

► **Lemma 5** (FJ strong determinism). *If $\mathcal{I}_{\text{FJ}} \vdash e \Downarrow v_1$ by a proof tree t_1 and $\mathcal{I}_{\text{FJ}} \vdash e \Downarrow v_2$ by a proof tree t_2 , then $t_1 = t_2$ and $v_1 = v_2$.*

Proof. By induction on the definition of $e \Downarrow v_1$. The key point is that each judgement is the consequence of exactly one rule. ◀

By relying on strong determinism, it is easy to see that in FJ a proof tree for an expression cannot contain another node labelled by the same expression. In other words, if the evaluation of e requires to evaluate e again, then the FJ semantics is undefined on e , as expected.

► **Lemma 6.** *A proof tree in \mathcal{I}_{FJ} for $e \Downarrow v$ cannot contain any other node $e \Downarrow v'$, for any v' .*

Proof. By Lemma 5, there is a unique proof tree t for the expression e . Hence, a node $e \Downarrow v'$ in t would be necessarily the root of a subtree of t equal to t , that is, it is the root of t . ◀

► **Definition 7.** *Let $\mathcal{I}_{\text{FJ}} \vdash e \Downarrow v$. A call trace τ is disjoint from $e \Downarrow v$ if in its proof tree⁹ there are no instances of (FJ-INVK) where $v_0.m(\bar{v}) \in \text{dom}(\tau)$.*

► **Lemma 8.** *If $\mathcal{I}_{\text{FJ}} \vdash e \Downarrow v$, then, for all τ disjoint from $e \Downarrow v$, we have $e, \emptyset, \tau \Downarrow v, \emptyset$.*

Proof. The proof is by induction on the definition of $e \Downarrow v$.

(FJ-field) Let τ be a call trace disjoint from $e.f \Downarrow v_i$. Since $\mathcal{I}_{\text{FJ}} \vdash e \Downarrow v$, with $v = \mathbf{new} C(v_1, \dots, v_n)$, holds by hypothesis, and τ is, by definition, also disjoint from $e \Downarrow v$, we get $e, \emptyset, \tau \Downarrow v, \emptyset$ by induction hypothesis. Then, since $\text{unfold}(v, \emptyset) = v$, we get $e.f, \emptyset, \tau \Downarrow v_i, \emptyset$ by rule (FIELD).

(FJ-new) Let τ be a call trace disjoint from $\mathbf{new} C(e_1, \dots, e_n) \Downarrow \mathbf{new} C(v_1, \dots, v_n)$. For all $i \in 1..n$, since $\mathcal{I}_{\text{FJ}} \vdash e_i \Downarrow v_i$ holds by hypothesis, and τ is, by definition, also disjoint from $e_i \Downarrow v_i$, we get $e_i, \emptyset, \tau \Downarrow v_i, \emptyset$ by induction hypothesis. Then, we get $\mathbf{new} C(e_1, \dots, e_n), \emptyset, \tau \Downarrow \mathbf{new} C(v_1, \dots, v_n), \emptyset$ by rule (NEW).

(FJ-invk) Let τ be a call trace disjoint from $e_0.m(e_1, \dots, e_n) \Downarrow v$. For all $i \in 0..n$, since $\mathcal{I}_{\text{FJ}} \vdash e_i \Downarrow v_i$ holds by hypothesis, and τ is, by definition, also disjoint from $e_i \Downarrow v_i$, we get $e_i, \emptyset, \tau \Downarrow v_i, \emptyset$ by induction hypothesis. Set $\bar{v} = v_1 \dots v_n$ and $e' = e[v_0/\mathbf{this}][\bar{v}/\bar{x}]$. By hypothesis, $\mathcal{I}_{\text{FJ}} \vdash e' \Downarrow v$ and, by definition, τ is also disjoint from $e' \Downarrow v$; furthermore, by Lemma 6, e' cannot occur twice in the proof tree for $e' \Downarrow v$, hence $\tau\{v_0.m(\bar{v}) : x\}$ is disjoint from $e' \Downarrow v$, for any fresh variable x . Then, by induction hypothesis, we have $e', \emptyset, \tau\{v_0.m(\bar{v}) : x\} \Downarrow v, \emptyset$, thus we get $e_0.m(e_1, \dots, e_n), \emptyset, \tau \Downarrow v, \emptyset$ by rule (INVK-OK). ◀

⁹ Unique thanks to Lemma 5.

We can now prove the conservativity result for COFJ operational semantics.

Proof of Theorem 4. The right-to-left direction follows from Lemma 8, since \emptyset is disjoint from any expression, while the other direction follows from the right-to-left one and Theorem 3. \blacktriangleleft

For COFJ operational semantics we can prove an additional result, characterizing derivable judgements which produce an empty environment. The meaning is that all results obtained *without using the codefinitions* are original FJ results.

► **Lemma 9.** *If $e, \emptyset, \tau \Downarrow v, \emptyset$ holds, then v is an FJ value v , and $\mathcal{I}_{FJ} \vdash e \Downarrow v$.*

5 Advanced examples

This section provides some more complex examples to better understand the operational semantics of COFJ in Section 4 and its relationship with the abstract semantics in Section 3.

Examples on lists. We first show an example motivating the additional checking step (third premise) in rule (INVK-CHECK). Essentially, the success of this check for some capsule result corresponds to the existence of an infinite tree in the abstract semantics, whereas the fact that this capsule result is obtained by assuming the codefinition as result of the cyclic call (second premise) corresponds to the existence of a finite tree which uses the codefinition.

Assume to add to our running example of lists of integers a method that returns the sum of the elements. For infinite regular lists, that is, lists ending with a cycle, a result should be returned if the cycle has sum 0, for instance for a list ending with infinitely many 0s, and no result if the cycle has sum different from 0. This can be achieved as follows.

```
class List extends Object { ...
  int sum() {0}
}
class NonEmptyList extends List { ...
  int sum() {this.head + this.tail.sum()} corec {0}
}
```

It is easy to see that the abstract semantics of the previous section formalizes the expected behavior. For instance, an infinite tree for a judgment $[2, 1]^\omega.\text{sum()} \Downarrow v$ only exists for $v = 2 + 1 + v$, and there are no solutions of this equation, hence there is no result. In the operational semantics, by evaluating the body assuming the codefinition as result of the cyclic call (second premise of rule (INVK-CHECK)) the spurious result 3 would be returned. This is avoided by the third premise, which evaluates the method body assuming 3 as result of the cyclic call. Since we *do not* get 3 in turn as result, evaluation is stuck, as expected.

Note that the stuckness situation is detected: the last side-condition of rule (INVK-CHECK) fails, and a dynamic error (not modeled for simplicity, see the comments to the rule) is raised, likely an exception in an implementation. On the other hand, computations which *never* reach (a base case or) an already encountered call still do not terminate in this operational semantics, exactly as in the standard one, and the fact that this does not happen should be *proved* by suitable techniques, see the Conclusion.

All the examples shown until now have a constant codefinition. We show now an example where this is not enough. Consider the method `remPos()` that removes positive elements. A first attempt at a COFJ definition is the following:

```

class NonEmptyList extends List { ...
  List remPos() {
    if(this.head > 0) this.tail.remPos()
    else new NonEmptyList(this.head,this.tail.remPos())}
  corec {new EmptyList()}
}

```

Is this definition correct? Actually, it provides the expected behavior on finite lists, and cyclic lists where the cycle contains only positive elements. However, when the cycle contains at least one non positive element, there is no result. For instance, consider the method call $[0, 1]^\omega.\text{remPos}()$. In the abstract semantics, an infinite tree can be constructed for the judgment $[0, 1]^\omega.\text{remPos}() \Downarrow v$ only if $v = 0 : v$, and this clearly only holds for $v = [0]^\omega$. However, no finite tree can be constructed for this judgment using the codefinition. Note that, in the operational semantics, without the additional check (third premise of rule (INVK-CHECK)), we would get the spurious result $[0]$. In order to have a COFJ definition complete with respect to the expected behavior, we should provide a different codefinition for lists with infinitely many non-positive elements.

```

class NonEmptyList extends List { ...
  List remPos() {
    if(this.head > 0) this.tail.remPos()
    else new NonEmptyList(this.head,this.tail.remPos())}
  corec {if (this.allPos() then new EmptyList() else any}
}

```

Arithmetic with rational and real numbers. All real numbers in the closed interval $\{0..1\}$ can be represented by infinite lists $[d_1, d_2, \dots]$ of decimal digits; more precisely, the infinite list $[d_1, d_2, \dots]$ represents the real number which is the limit of the series $\sum_{i=1}^{\infty} 10^{-i} d_i$.

It is well-known that all rational numbers in $\{0..1\}$ correspond to either a terminating or repeating decimal, hence they can be represented by infinite regular lists of digits, where terminating decimals end with either an infinite sequence of 0 or an infinite sequence of 9; for instance, the terminating decimal $\frac{1}{2}$ can be represented equivalently by either $[5, 0, 0, \dots]$ or $[4, 9, 9, \dots]$, while the repeating decimal $\frac{1}{3}$ is represented by $[3, 3, \dots]$.

Therefore, in COFJ all rational numbers in $\{0..1\}$ can be effectively represented with infinite precision at the level of the operational semantics; to this aim, we can declare a class `Number` with the two fields `digit` of type `int` and `others` of type `Number`: `digit` contains the leftmost digit, that is, the most significant, while `others` refers to the remaining digits, that is, the number we would obtain by a single left shift (corresponding to multiplication by 10). Since also non-regular values are allowed, in the abstract semantics class `Number` can be used to represent also all irrational numbers in $\{0..1\}$.

We now show how it is possible to compute in COFJ the addition of rational numbers in $\{0..1\}$ with infinite precision. We first define the method `carry` which computes the carry of the addition of two numbers: its result is 0 if the sum belongs to $\{0..1\}$, 1 otherwise.

```

class Number extends Object { // numbers in {0..1}
  int digit; // leftmost digit
  Number others; // all other digits

  int carry(Number num){ // returns 0 if this+num<=1, 1 otherwise
    if (this.digit+num.digit!=9) (this.digit+num.digit)/10
    else this.others.carry(num.others)
  } corec {0}
}

```

1:18 Sound Regular Corecursion in coFJ

The two numbers `this` and `num` are inspected starting from the most significant digits: if their sum is different from 9, then the carry can be computed without inspecting the other digits, hence the integer division by 10 of the sum is returned. Corecursion is needed when the sum of the two digits equals 9; in this case the carry is the same obtained from the addition of `this.others` and `num.others`.

Finally, in the codefinition the carry 0 is returned; indeed, the codefinition is evaluated only when the sum of the digits for all positions inspected so far is 9 and the same patterns of digits are encountered for the second time. This can only happen for pairs of numbers whose addition is $[9, 9, \dots]$, that is, 1, hence the computed carry must be 0.

Based on method `carry`, we can define method `add` which computes the addition of two numbers, excluding the possible carry in case of overflow.

```
class Number extends Object { ... // declarations as above
  Number add(Number num){ // returns this+num
    new Number(
      (this.digit+num.digit+this.others.carry(num.others))%10,
      this.others.add(num.others))} corec {any}
}
```

For each position, the corresponding digits of `this` and `num` are added to the carry computed for the other digits (`this.others.carry(num.others)`), then the remainder of the division by 10 gives the most significant digit of the result, whereas the others are obtained by corecursively calling the method on the remaining digits (`this.others.add(num.others)`). Since this call is guarded by a constructor call, the codefinition is `any`.

Note that, in the abstract semantics, methods `carry` and `add` correctly work also for irrational numbers.

Method `add` above is simple, but has the drawback that the same carries are computed more times; hence, in the worst case, the time complexity is quadratic in the period¹⁰ of the two involved repeating decimals. To overcome this issue, we present a more elaborate example where carries are computed only once for any position; this is achieved by method `all_carries` below, which returns the sequence of all carries (hence, a list of binary digits).

Method `simple_add` corecursively adds all digits without considering carries, while method `add`, defined on top of `simple_add` and `all_carries`, computes the final result. This new version of `add` is not recursive and, hence, does not need a codefinition.

```
class Number extends Object { ... // declarations as above
  Number all_carries(Number num){ // carries for all positions
    this.simple_carries(num).complete()
  }
  Number simple_carries(Number num){ // carries computed immediately
    if(this.digit+num.digit!=9)
      new Number((this.digit+num.digit)/10,
        this.others.simple_carries(num.others))
    else new Number(9, this.others.simple_carries(num.others))
  } corec {any}

  Number complete(){ // computes missing carries marked with 9
    if(this.digit!=9) new Number(this.digit, this.others.complete())
    else this.fill(this.carry_lookahead()).complete()
  } corec {any}
}
```

¹⁰ Indeed, the worst case scenario is when the carry propagates over all digits because their sum is always 9, and this can happen only if the two numbers have the same period.

```

Number fill(int dig){ // fills with dig all next missing carries
  if(this.digit!=9) this else new Number(dig,this.others.fill(dig))
} corec {any}

int carry_lookahead(){ // returns the next computed carry
  if(this.digit!=9) this.digit else this.others.carry_lookahead()
} corec {0}

Number simple_add(Number num){ // addition without carries
  new Number((this.digit+num.digit)%10,
    this.others.simple_add(num.others))
} corec {any}

Number add(Number num){
  this.simple_add(num).simple_add(this.all_carries(num).others)
}
}

```

Distances on graphs. The last example of this section involves graphs, which are the paradigmatic example of cyclic data structure. Our aim is to compute the *distance*, that is, the minimal length of a path, between two vertexes¹¹. Consider a graph (V, adj) where V is the set of vertexes and $adj : V \rightarrow \wp(V)$ gives, for each vertex, the set of the adjacent vertexes. Each vertex has an identifier *id* assumed to be unique. We assume a class Nat^∞ , with subclasses *Nat* with an integer field, and *Infty* with no fields, for naturals and ∞ (distance between unconnected nodes), respectively. Such classes offer methods *succ()* for the successor, and *min(Nat[∞] n)* for the minimum, with the expected behaviour (e.g., *succ* in class Nat^∞ returns ∞).

```

class Vertex extends Object {
  Id id; AdjList adjVerts;
  Nat∞dist(Id id) {
    this.id==id?new Nat(0):this.adjVerts.dist(id).succ()}
  corec {new Infty()}
}

class AdjList extends Object { }
class EAdjList extends AdjList {
  Nat∞dist(Id id) { new Infty() }
}
class NEAdjList extends AdjList {
  Vertex vert; AdjList adjVerts;
  Nat∞dist(Id id) {this.vert.dist(id).min(this.adjVerts.dist(id))}
}

```

Clearly, if the destination *id* and the source node coincide, then the distance is 0. Otherwise, the distance is obtained by incrementing by one the minimal distance from an adjacent to *id*, computed by method *dist()* of *AdjList* called on the adjacency list. The codefinition of method *dist()* of class *Vertex* is needed since, in presence of a cycle, ∞ is returned and non-termination is avoided. The same approach can be adopted for visiting a graph: instead of keeping trace of already encountered nodes, cycles are implicitly handled by the loop detection mechanism of COFJ.

¹¹The example can be easily adapted to weighted paths.

6 Soundness

Soundness of the operational semantics with respect to the abstract one means, roughly, that a value derived using the rules in Figure 6 can also be derived by those in Figure 2. However, this statement needs to be refined, since values in the two semantics are different: possibly infinite objects in the abstract semantics, and capsules in the operational semantics.

We define a relation from capsules to abstract objects, formally express soundness through this relation, and introduce an intermediate semantics to carry out the proof in two steps.

From capsules to infinite objects. Intuitively, given a capsule (v, σ) , we get an abstract value by instantiating variables in v with abstract values, in a way consistent with σ . To make this formal, we need some preliminary definitions.

A *substitution* θ is a function from variables to abstract values. We denote by $e\theta$ the abstract expression obtained by applying θ to e . In particular, if e is an open value v , then $v\theta$ is an abstract value. Given an environment σ and a substitution θ , the substitution $\sigma[\theta]$ is defined by:

$$\sigma[\theta](x) = \begin{cases} \sigma(x)\theta & x \in \text{dom}(\sigma) \\ \theta(x) & x \notin \text{dom}(\sigma) \end{cases}$$

Then, a *solution* of σ is a substitution θ such that $\sigma[\theta] = \theta$. Let $\text{Sol}(\sigma)$ be the set of solutions of σ . Finally, if (e, σ) is a capsule, we define the set of abstract expressions it denotes as $\llbracket e, \sigma \rrbracket = \{e\theta \mid \theta \in \text{Sol}(\sigma)\}$. Note that $\llbracket v, \sigma \rrbracket \subseteq \mathcal{V}^a$, for any capsule (v, σ) . We now show an operational characterization of the semantic equality.

► **Theorem 10.** $\llbracket v_1, \sigma_1 \rrbracket = \llbracket v_2, \sigma_2 \rrbracket$ iff $(v_1, \sigma_1) \approx_\alpha (v_2, \sigma_2)$, for some σ_1, σ_2 -renaming α .

To prove this result we need some auxiliary definitions and lemmas. The *tree expansion* of a capsule (v, σ) is the possibly infinite open value coinductively defined as follows:

$$T(v, \sigma) = \begin{cases} x & v = x \text{ and } \text{unfold}(x, \sigma) \uparrow \\ \mathbf{new} C(T(v_1, \sigma), \dots, T(v_n, \sigma)) & \text{unfold}(v, \sigma) = \mathbf{new} C(v_1, \dots, v_n) \end{cases}$$

The next proposition shows relations between solutions and tree expansion of a capsule.

► **Proposition 11.** Let (v, σ) be a capsule and $\theta \in \text{Sol}(\sigma)$, then

1. if $\text{unfold}(v, \sigma) \uparrow$ then $v = x$ and $x \overset{\sigma}{\leftrightarrow} x$
2. $FV(T(v, \sigma)) \subseteq \{x \in \text{dom}(\sigma) \mid x \overset{\sigma}{\leftrightarrow} x\}$
3. if $x \overset{\sigma}{\leftrightarrow} y$ then $\theta(x) = \theta(y)$
4. if $\text{unfold}(v, \sigma) = \mathbf{new} C(v_1, \dots, v_n)$ then $v\theta = \mathbf{new} C(v_1\theta, \dots, v_n\theta)$
5. $v\theta = T(v, \sigma)\theta$

Given a relation α on variables, we will denote by α° the opposite relation and by $=_\alpha$ the equality of possibly infinite open values up-to α , coinductively defined by the following rules:

$$\frac{}{x =_\alpha y} \quad x\alpha y \quad \frac{t_i =_\alpha s_i \quad \forall i \in 1..n}{\mathbf{new} C(t_1, \dots, t_n) =_\alpha \mathbf{new} C(s_1, \dots, s_n)}$$

It is easy to check that

- α is a σ_1, σ_2 -renaming iff α° is a σ_2, σ_1 -renaming,
- $(v_1, \sigma_1) \approx_\alpha (v_2, \sigma_2)$ iff $(v_2, \sigma_2) \approx_{\alpha^\circ} (v_1, \sigma_1)$,
- $t_1 =_\alpha t_2$ iff $t_2 =_{\alpha^\circ} t_1$.

We have the following lemmas:

► **Lemma 12.** $(v_1, \sigma_1) \approx_\alpha (v_2, \sigma_2)$ iff $T(v_1, \sigma_1) =_\alpha T(v_2, \sigma_2)$, for each σ_1, σ_2 -renaming α .

Proof. The proof is immediate by coinduction in both directions. ◀

► **Lemma 13.** If $T(v_1, \sigma_1) =_\alpha T(v_2, \sigma_2)$, where α is a σ_1, σ_2 -renaming, then $\llbracket v_1, \sigma_1 \rrbracket = \llbracket v_2, \sigma_2 \rrbracket$.

► **Proposition 14.** If $\llbracket v_1, \sigma_1 \rrbracket = \llbracket v_2, \sigma_2 \rrbracket$ then

1. if $\text{unfold}(v_1, \sigma_1) \uparrow$ then $\text{unfold}(v_2, \sigma_2) \uparrow$,
2. if $\text{unfold}(v_1, \sigma_1) = \text{new } C(v_{1,1}, \dots, v_{1,n})$ then $\text{unfold}(v_2, \sigma_2) = \text{new } C(v_{2,1}, \dots, v_{2,n})$ and, for all $i \in 1..n$, $\llbracket v_{1,i}, \sigma_1 \rrbracket = \llbracket v_{2,i}, \sigma_2 \rrbracket$.

► **Lemma 15.** If $\llbracket v_1, \sigma_1 \rrbracket = \llbracket v_2, \sigma_2 \rrbracket$ then $T(v_1, \sigma_1) =_\alpha T(v_2, \sigma_2)$, for some σ_1, σ_2 -renaming α .

Proof of Theorem 10. The right-to-left direction follows from Lemma 12 and Lemma 13, while the other direction follows from Lemma 15 and Lemma 12. ◀

Since by definition \approx is equal to \approx_α for some α , applying Lemma 12 and Lemma 13 we get that if $(v_1, \sigma_1) \approx (v_2, \sigma_2)$ then $\llbracket v_1, \sigma_1 \rrbracket = \llbracket v_2, \sigma_2 \rrbracket$. Actually we can prove a stronger result:

► **Lemma 16.** If $(v_1, \sigma_1) \approx_\alpha (v_2, \sigma_2)$ for some strict σ_1, σ_2 -renaming α , then, for each solution $\theta \in \text{Sol}(\sigma_1 \cap \sigma_2)$, there are $\theta_1 \in \text{Sol}(\sigma_1)$ and $\theta_2 \in \text{Sol}(\sigma_2)$ such that $v_1 \theta_1 = v_2 \theta_2$ and, for all $x \in \text{dom}(\sigma_1 \cap \sigma_2)$, $\theta_1(x) = \theta(x) = \theta_2(x)$.

Soundness statement. We can now formally state the soundness result:

► **Theorem 17.** If $e, \emptyset, \emptyset \Downarrow v, \sigma$, then, for all $v \in \llbracket v, \sigma \rrbracket$, $(\mathcal{I}_{FJ}, \mathcal{I}_{FJ}^{co}) \vdash e \Downarrow v$.

This main result is about the evaluation of source expressions, hence both the environment and the call trace are empty. To carry out the proof we need to generalize the statement.

► **Theorem 18 (Soundness).** If $e, \sigma, \emptyset \Downarrow v, \sigma'$, then, for all $\theta \in \text{Sol}(\sigma')$, $(\mathcal{I}_{FJ}, \mathcal{I}_{FJ}^{co}) \vdash e \theta \Downarrow v \theta$.

To show that this is actually a generalization, set $\sigma_1 \leq \sigma_2$ if $\text{dom}(\sigma_1) \subseteq \text{dom}(\sigma_2)$, and, for all $x \in \text{dom}(\sigma_1)$, $\sigma_1(x) = \sigma_2(x)$. We use the following lemmas.

► **Lemma 19.** If $\sigma_1 \leq \sigma_2$, then $\text{Sol}(\sigma_2) \subseteq \text{Sol}(\sigma_1)$.

► **Lemma 20.** If $e, \sigma, \tau \Downarrow v, \sigma'$, then $\sigma \leq \sigma'$.

In the statement of Theorem 18, thanks to Lemma 20, we know that $\sigma \leq \sigma'$, hence, by Lemma 19, $\theta \in \text{Sol}(\sigma)$, thus $e \theta \in \llbracket e, \sigma \rrbracket$. Theorem 18 implies Theorem 17, since, when $\sigma = \emptyset$, e is closed, hence $e \theta = e$, and all elements in $\llbracket v, \sigma' \rrbracket$ have shape $v \theta$ with $\theta \in \text{Sol}(\sigma')$.

Proof through intermediate semantics. In order to prove Theorem 18, we introduce a new semantics called *intermediate*, defined in Figure 8. Values are those of the abstract semantics, hence calls are of shape $v.m(\bar{v})$ (*abstract calls*). The judgment has shape $e, \rho, S \Downarrow_{\text{IN}} v, S'$, with S, S' sets of abstract calls, ρ map from abstract calls to values. Comparing with $e, \sigma, \tau \Downarrow v, \sigma'$ in the operational semantics, no variables are introduced for calls; ρ and S play the role of the ck and non ck part of τ , respectively, keeping trace of already encountered calls. Moreover, ρ directly associates to a call its value to be used in the checking step, which in σ is associated to the corresponding variable. Finally, S' plays the role of σ' , tracing the calls for which the codefinition has been evaluated, hence the checking step will be needed. This correspondence is made precise below. The rules are analogous to those of Figure 6, with the difference that,

for an already encountered call $c \in S$, either rule (IN-INVK-OK) or rule (IN-COREC) can be applied. In other words, evaluation of the codefinition is not necessarily triggered when the *first* cycle is detected. This non-determinism makes the relation with the abstract semantics simpler.

By relying on the intermediate semantics, we can prove Theorem 18 by two steps:

1. The operational semantics is sound w.r.t. the intermediate semantics (Theorem 21).
2. The intermediate semantics is sound w.r.t. the abstract semantics (Theorem 23).

At the beginning of Section 4, we mentioned two issues for an operational semantics: representing infinite objects in a finite way, and replacing infinite (regular) proof trees by finite proof trees. This proof technique nicely shows that the two issues are orthogonal: notably, detection of cyclic calls is independent from the format of values.

To express the soundness of the operational semantics w.r.t. the intermediate one, we need to formally relate the two judgments. First of all, a call trace τ is the disjoint union of two maps τ^{ck} and $\tau^{-\text{ck}}$ into tagged and non-tagged variables, respectively. Then, given an environment σ , we define the following sets of (operational) calls:

- $S^\tau = \text{dom}(\tau^{-\text{ck}})$
- $S^{\tau, \sigma} = \text{dom}(\sigma \circ \tau^{-\text{ck}})$, where \circ is the composition of partial functions
- $S^{\tau, \sigma, \sigma'} = S^{\tau, \sigma'} \setminus S^{\tau, \sigma}$

For S set of calls and θ substitution, we abbreviate by S_θ the set of abstract calls $S\theta$. Note that $S_\theta^{\tau, \sigma} \subseteq S_\theta^\tau$ and, if $\sigma_1 \leq \sigma_2$, then $S_\theta^{\tau, \sigma_1} \subseteq S_\theta^{\tau, \sigma_2}$. Finally, $\rho_\theta^\tau(c\theta) = v$ iff $v = \theta(\tau^{\text{ck}}(c))$.

Then, the soundness result can be stated as follows:

► **Theorem 21** (Soundness operational w.r.t. intermediate). *If $e, \sigma, \tau \Downarrow v, \sigma'$ then, for all $\theta \in \text{Sol}(\sigma')$, there exists S such that $S_\theta^{\tau, \sigma, \sigma'} \subseteq S \subseteq S_\theta^{\tau, \sigma'}$ and, $e\theta, \rho_\theta^\tau, S_\theta^\tau \Downarrow_{\text{IN}} v\theta, S$.*

In particular, the bounds on S ensure that it is empty when $\tau = \emptyset$. Hence, if $e, \sigma, \emptyset \Downarrow v, \sigma'$ (hypothesis of Theorem 18), then $e\theta, \emptyset, \emptyset \Downarrow_{\text{IN}} v\theta, \emptyset$, that is, the hypothesis of Theorem 23 below holds.

The proof of the theorem uses the following corollary of Lemma 16.

► **Corollary 22.** *If $(v_1, \sigma_1) \approx (v_2, \sigma_2)$, $\theta_1 \in \text{Sol}(\sigma_1)$, $\sigma_1 \leq \sigma_2$, then there is $\theta_2 \in \text{Sol}(\sigma_2)$ such that $v_1\theta_1 = v_2\theta_2$ and, for all $x \in \text{dom}(\sigma_1)$, $\theta_1(x) = \theta_2(x)$. Moreover, if $\sigma_1 = \sigma_2$, then $v_1\theta_1 = v_2\theta_1$.*

We now state the second step of the proof: the soundness result of the intermediate semantics with respect to the abstract semantics.

► **Theorem 23** (Soundness intermediate w.r.t. abstract). *If $e, \emptyset, \emptyset \Downarrow_{\text{IN}} v, \emptyset$, then $(\mathcal{I}_{\text{FJ}}, \mathcal{I}_{\text{FJ}}^{\text{co}}) \vdash e \Downarrow v$.*

The proof uses the bounded coinduction principle (Theorem 1), and requires some lemmas. Recall that $\mathcal{I}_{\text{FJ}} \cup \mathcal{I}_{\text{FJ}}^{\text{co}} \vdash e \Downarrow v$ means that the judgment $e \Downarrow v$ has a finite proof tree in the (standard) inference system consisting of FJ rules and coFJ corules.

► **Lemma 24.** *If $e, \emptyset, S \Downarrow_{\text{IN}} v, S'$ then $\mathcal{I}_{\text{FJ}} \cup \mathcal{I}_{\text{FJ}}^{\text{co}} \vdash e \Downarrow v$ holds.*

► **Lemma 25.** *If $e, \rho, S \cup \{c\} \Downarrow_{\text{IN}} v, S'$ holds, and $c \notin S'$, then $e, \rho, S \Downarrow_{\text{IN}} v, S'$.*

► **Lemma 26.** *If $e, \rho\{c : v'\}, S \Downarrow_{\text{IN}} v, S'$ and $c, \rho, S \Downarrow_{\text{IN}} v', \emptyset$, then $e, \rho, S \Downarrow_{\text{IN}} v, S'$.*

We can now prove Theorem 23.

Proof of Theorem 23. We take as specification the set $A = \{(e, v) \mid e, \emptyset, \emptyset \Downarrow_{\text{IN}} v, \emptyset\}$, and we use bounded coinduction (Theorem 1). We have to prove the following:

$v \in \mathcal{V}^a$	$::=_{\text{CO}}$	$\text{new } C(\bar{v})$	possibly infinite object
c	$::=$	$v.m(\bar{v})$	abstract call
S	$::=$	$c_1 \dots c_n \quad (n \geq 0)$	set of abstract calls
ρ	$::=$	$c_1 : v_1 \dots c_n : v_n \quad (n \geq 0)$	

$$\begin{array}{c} \text{(IN-VAL)} \frac{}{v, \rho, S \Downarrow_{\text{IN}} v, \emptyset} \quad \text{(IN-FIELD)} \frac{e, \rho, S \Downarrow_{\text{IN}} v, S' \quad v = \text{new } C(v_1, \dots, v_n)}{e.f, \rho, S \Downarrow_{\text{IN}} v_i, S'} \quad \begin{array}{l} \text{fields}(C) = f_1 \dots f_n \\ f = f_i, i \in 1..n \end{array} \end{array}$$

$$\text{(IN-NEW)} \frac{e_i, \rho, S \Downarrow_{\text{IN}} v_i, S'_i \quad \forall i \in 1..n}{\text{new } C(e_1, \dots, e_n), \rho, S \Downarrow_{\text{IN}} \text{new } C(v_1, \dots, v_n), \bigcup_{i \in 1..n} S'_i}$$

In all the following rules:

$$\begin{array}{l} \bar{e} = e_1, \dots, e_n \\ \bar{v} = v_1 \dots v_n \\ c = v_0.m(\bar{v}) \\ v_0 = \text{new } C(_) \end{array}$$

$$\text{(IN-INVOK-OK)} \frac{e_i, \rho, S \Downarrow_{\text{IN}} v_i, S'_i \quad \forall i \in 0..n \quad e[v_0/\text{this}][\bar{v}/\bar{x}], \rho, S \cup \{c\} \Downarrow_{\text{IN}} v, S'}{e_0.m(\bar{e}), \rho, S \Downarrow_{\text{IN}} v, \bigcup_{i \in 0..n} S'_i \cup S'} \quad \begin{array}{l} c \notin S' \text{ or } c \in S \\ \text{mbody}(C, m) = (\bar{x}, e) \end{array}$$

$$\text{(IN-INVOK-CHECK)} \frac{e_i, \rho, S \Downarrow_{\text{IN}} v_i, S'_i \quad \forall i \in 0..n \quad e[v_0/\text{this}][\bar{v}/\bar{x}], \rho, S \cup \{c\} \Downarrow_{\text{IN}} v, S' \quad e[v_0/\text{this}][\bar{v}/\bar{x}], \rho\{c:v\}, S \Downarrow_{\text{IN}} v, S''}{e_0.m(\bar{e}), \rho, S \Downarrow_{\text{IN}} v, \bigcup_{i \in 0..n} S'_i \cup (S' \setminus \{c\})} \quad \begin{array}{l} c \notin S \\ \text{mbody}(C, m) = (\bar{x}, e) \\ c \in S' \end{array}$$

$$\text{(IN-COREC)} \frac{e_i, \rho, S \Downarrow_{\text{IN}} v_i, S'_i \quad \forall i \in 0..n \quad e'[v_0/\text{this}][\bar{v}/\bar{x}][u/\text{any}], \rho, S \Downarrow_{\text{IN}} v, S'}{e_0.m(\bar{e}), \rho, S \Downarrow_{\text{IN}} v, \bigcup_{i \in 0..n} S'_i \cup S' \cup \{c\}} \quad \begin{array}{l} c \in S \\ \text{co-mbody}(C, m) = (\bar{x}, e') \\ c \notin \text{dom}(\rho) \end{array}$$

$$\text{(IN-LOOK-UP)} \frac{e_i, \rho, S \Downarrow_{\text{IN}} v_i, S'_i \quad \forall i \in 0..n}{e_0.m(\bar{e}), \rho, S \Downarrow_{\text{IN}} v, \bigcup_{i \in 0..n} S'_i} \quad \rho(c) = v$$

■ **Figure 8** COFJ intermediate semantics.

Boundedness For all $(e, v) \in A$, $\mathcal{I}_{FJ} \cup \mathcal{I}_{FJ}^{co} \vdash e \Downarrow v$ holds.

Consistency For all $(e, v) \in A$, there exist a rule in the abstract semantics having $e \Downarrow v$ as consequence, and such that all its premises are elements of A .

Boundedness follows immediately from Lemma 24. We now prove consistency.

Consider a pair $(e, v) \in A$, hence we know that $e, \emptyset, \emptyset \Downarrow_{IN} v, \emptyset$ is derivable. We proceed by case analysis on the last applied rule in the derivation of this judgement.

(IN-val) We know that $e = v = \mathbf{new} C(v_1, \dots, v_n)$. We choose as candidate rule (ABS-NEW).

We have to show that, for all $i \in 1..n$, $(v_i, v_i) \in A$, that is, $v_i, \emptyset, \emptyset \Downarrow_{IN} v_i, \emptyset$ holds. We can get the thesis thanks to rule (IN-VAL).

(IN-field) We know that $e = e'.f$ and $e', \emptyset, \emptyset \Downarrow_{IN} \mathbf{new} C(v_1 \dots v_n), \emptyset$. We choose as candidate rule (ABS-FIELD), with conclusion $e'.f \Downarrow v_i$. We have to show that $(e', \mathbf{new} C(v_1 \dots v_n)) \in A$, that is, $e', \emptyset, \emptyset \Downarrow_{IN} \mathbf{new} C(v_1 \dots v_n), \emptyset$ holds, but this is true by hypothesis.

(IN-new) We know that $e_i, \emptyset, \emptyset \Downarrow_{IN} v_i, \emptyset$ holds for all $i \in 1..n$. We choose as candidate rule (ABS-NEW). We have to show that, for all $i \in 1..n$, $(e_i, v_i) \in A$, that is, $e_i, \emptyset, \emptyset \Downarrow_{IN} v_i, \emptyset$ holds, but this is true by hypothesis.

(IN-invk-ok) We know that $e = e_0.m(\bar{e})$, $e_i, \emptyset, \emptyset \Downarrow_{IN} v_i, \emptyset$ holds for all $i \in 0..n$, $c = v_0.m(\bar{v})$, $mbody(C, m) = (\bar{x}, e')$, and $e'[v_0/\mathbf{this}][\bar{v}/\bar{x}], \emptyset, \{c\} \Downarrow_{IN} v, \emptyset$ holds. We choose as candidate rule (ABS-INVK). We have to show that, for all $i \in 0..n$, $(e_i, v_i) \in A$, and $(e'[v_0/\mathbf{this}][\bar{v}/\bar{x}], v) \in A$. That is, that the following judgments hold: $e_i, \emptyset, \emptyset \Downarrow_{IN} v_i, \emptyset$ for all $i \in 0..n$, and $e'[v_0/\mathbf{this}][\bar{v}/\bar{x}], \emptyset, \emptyset \Downarrow_{IN} v, \emptyset$. The judgments in the first set hold by hypothesis. The last judgment holds thanks to Lemma 25, where $S' = \emptyset$.

(IN-invk-check) We know that $e = e_0.m(\bar{e})$, $e_i, \emptyset, \emptyset \Downarrow_{IN} v_i, \emptyset$ holds for all $i \in 0..n$, $c = v_0.m(\bar{v})$, $mbody(C, m) = (\bar{x}, e')$, and $e'[v_0/\mathbf{this}][\bar{v}/\bar{x}], \{c : v\}, \emptyset \Downarrow_{IN} v, \emptyset$ holds. We choose as candidate rule (ABS-INVK). We have to show that for all $i \in 0..n$, $(e_i, v_i) \in A$, and $(e'[v_0/\mathbf{this}][\bar{v}/\bar{x}], v) \in A$. That is, that the following judgments hold: $e_i, \emptyset, \emptyset \Downarrow_{IN} v_i, \emptyset$ for all $i \in 0..n$, and $e'[v_0/\mathbf{this}][\bar{v}/\bar{x}], \emptyset, \emptyset \Downarrow_{IN} v, \emptyset$. The judgments in the first set hold by hypothesis. The last judgment holds thanks to Lemma 26, since from the hypothesis we easily get $c, \emptyset, \emptyset \Downarrow_{IN} v, \emptyset$.

(IN-corec) Empty case since to apply the rule it should be $S \neq \emptyset$.

(IN-look-up) Empty case since to apply the rule it should be $\rho \neq \emptyset$.

◀

7 Related work

As already mentioned, the idea of regular corecursion (keeping track of pending method calls, so to detect cyclic calls), originates from co-*SLD* resolution [20, 21, 7]. Making regular corecursion *flexible* means that the programmer can specify the behaviour in case a cycle is detected. Language constructs to achieve such flexibility have been proposed in the logic [2, 3], functional [17], and object-oriented [8, 9] paradigm.

Logic paradigm. The above mentioned *co-*SLD* resolution* [20, 21, 7] is a sound resolution procedure based on cycle detection. That is, the interpreter keeps track of resolved atoms and an atom selected from the current goal can be resolved if it unifies with an atom that has been already resolved. In this way it is possible to define inductive predicates. Correspondingly, models are subsets of the *complete Herbrand basis*, that is, the set of ground atoms built on arbitrary (finite or infinite) terms, and the declarative semantics is the greatest fixed point of the monotone function associated with a program. Structural resolution [18, 14] (a.k.a. S-resolution) is a proposed generalization for cases when formulas computable at infinity are

not regular; infinite derivations that cannot be built in finite time are generated lazily, and only partial answers are shown. More recently, a comprehensive theory has been proposed [11] to provide operational semantics that go beyond loop detection.

Anyway, in coinductive logic programming, only standard coinduction is supported. The notion of `finally` clause, introduced in [2], allows the programmer to specify a fact to be resolved when a cycle is detected, instead of simply accepting the atom. The approach has been refined in [3], following the guidelines given by the formal framework of generalized inference systems. That is, the programmer can write special clauses corresponding to corules, so that, when an atom is found for the second time, standard SLD resolution is triggered in the program enriched by the corules. However, this paradigm is very different from the object-oriented one, since based on relations rather than functions: cycles are detected on the same atom, where input and output are not distinguished, by unification.

Functional paradigm. *CoCaml* (www.cs.cornell.edu/Projects/CoCaml) [17, 16] is a fully-fledged extension of OCaml supporting non-well-founded data types and corecursive functions. CoCaml, as OCaml, allows programmers to declare regular values through the `let-rec` construct, and, moreover, detects cyclic calls as in our approach. However, whereas COFJ immediately evaluates the cyclic call by using the codefinition, the CoCaml approach is in two phases. First, a system of equations is constructed, associating with each call a variable and partially evaluating the body of functions, where calls are replaced with associated variables. Then, the system of equations is given to a *solver* specified in the function definition. Solvers can be either pre-defined or written by the programmer in order to enhance flexibility. An advantage that we see in our approach is that the programmer has to write the codefinition (standard code) rather than working at the meta-level to write a solver, which is in a sense a fragment of the interpreter. A precise comparison is difficult for the lack of a simple operational model of the CoCaml mechanism. In future work, we plan to develop such model, and to relate the two approaches on a formal basis.

Object-oriented paradigm. A previous version of COFJ has been proposed in [8]. At this time, however, the framework of inference systems with corules was still to come, so there was no formal model against which to check the given operational semantics, which, indeed, derived spurious results in some cases, as illustrated in Section 4 at page 16. The operational semantics provided in the current paper solves this problem, and is proved to be sound with respect to the abstract semantics. Moreover, we adopt a simpler representation of cyclic objects through capsules [16]. A type system has been proposed [9] for the previous version of COFJ to prevent *unsafe* use of the “undetermined” value. We leave to further work the investigation of typing issues for the approach presented in this paper.

8 Conclusion

The Java-like calculus presented in this paper promotes a novel programming style, which smoothly incorporates support for cyclic data structures and coinductive reasoning, in the object-oriented paradigm. Our contribution is foundational: we provide an abstract semantics based on corules and show that it is possible to define a *sound* operational model; such operational semantics is inductive, syntax-directed and deterministic, hence can be directly turned into an interpreter. In order to get a “real-world” language, of course many other issues should be taken into account.

Our prototype implements the *abstract* semantics on top of a Prolog meta-interpreter supporting flexible regular corecursion [3]. In this way, the inference system is naturally translated in Prolog¹², cyclic terms are natively supported, and their equality handled by unification. A fully-fledged interpreter of the *operational* semantics should directly handle these issues and, moreover, attempt at some optimization.

The current paper does not deal with types: an important concern is to guarantee *type soundness*, statically ensuring that an undetermined value never occurs as receiver of field access or method invocation, as investigated in [9] for the previous COFJ version [8].

Another issue is how to train developers to write codedefinitions. Standard recursion is non-trivial as well for beginners, whereas it becomes quite natural after understanding its mechanism. For regular corecursion the same holds, with is the additional difficulty of reasoning on infinite structures. Intuitively, the codedefinition can be regarded as a base case to be applied when a loop is detected. Moreover, again as for standard recursion, this novel programming style could be integrated with proof techniques to show the correctness of algorithms on cyclic data structures. Such proofs could be mechanized in proof assistants, as Agda, that provide built-in support for coinductive definitions and proofs by coinduction.

Finally, a non-trivial challenge is how to integrate regular corecursion, requiring to detect “the same call”, with the notion of mutable state. Likely, some immutability constraints will be needed, or a variant of the model where such a check requires a stateless computation. Another solution is to consider the check as an assertion that can be disabled if the programmer has verified the correctness of the method by hand or assisted by a tool.

The semantics of flexible regular corecursion in the paper is the operational counterpart of that obtained by considering recursive functions as relations, and recursive definitions (with codedefinition) as inference systems (with corules). We prove that the operational semantics is *sound* with respect to that interpretation. Obviously, *completeness* does not hold in general, since the abstract semantics deals with not only cyclic data structures (such as $[2, 1]^\omega$), but arbitrary non-well-founded structures (such as the list of natural numbers). Even considering only regular proof trees in the abstract semantics, in some subtle cases there is more than one admissible result¹³, whereas the operational semantics, being deterministic, finds “the first” among such results, as reasonable in an implementation. We plan to investigate such completeness issues in further work, also in the more general framework of inference systems, that is, to characterize judgments which have a regular proof tree.

We also plan to study how to deal with flexible corecursion in other programming paradigms, notably in the functional paradigm, and to compare on a formal basis this approach with the CoCaml approach relying on solvers, rather than codedefinitions.

As already discussed in the Introduction, lazy evaluation and regular corecursion are complementary approaches to deal with infinite data structures. With the lazy approach, arbitrary (computable) non-well-founded data structures are supported. However, we cannot compute results which need to explore the whole structure, whereas, with regular corecursion, this becomes possible for cyclic structures: for instance we can compute `allPos one_two`, which diverges in Haskell. A natural question is then whether it is possible to extend the regular corecursion approach to manage also non-regular objects, thus overcoming the principal drawback with respect to the lazy approach. A possible interesting direction, exploiting the work of Courcelle [12] on infinite trees, could be to move from regular to *algebraic* objects.

¹² A logic program can be seen as an inference system where judgments are atoms.

¹³ For instance, the list with no repetitions extracted from $[1, 2]^\omega$ can be either $[1, 2]$ or $[2, 1]$.

References

- 1 P. Aczel. An introduction to inductive definitions. In *Handbook of Mathematical logic*. North Holland, 1977.
- 2 Davide Ancona. Regular corecursion in Prolog. *Computer Languages, Systems & Structures*, 39(4):142–162, 2013.
- 3 Davide Ancona, Francesco Dagnino, and Elena Zucca. Extending coinductive logic programming with co-facts. In Ekaterina Komendantskaya and John Power, editors, *First Workshop on Coalgebra, Horn Clause Logic Programming and Types, CoALP-Ty’16*, volume 258 of *Electronic Proceedings in Theoretical Computer Science*, pages 1–18. Open Publishing Association, 2017. doi:10.4204/EPTCS.258.1.
- 4 Davide Ancona, Francesco Dagnino, and Elena Zucca. Generalizing inference systems by coaxioms. In Hongseok Yang, editor, *26th European Symposium on Programming, ESOP 2017*, volume 10201 of *Lecture Notes in Computer Science*, pages 29–55. Springer, 2017. doi:10.1007/978-3-662-54434-1_2.
- 5 Davide Ancona, Francesco Dagnino, and Elena Zucca. Reasoning on divergent computations with coaxioms. *PACMPL*, 1(OOPSLA):81:1–81:26, 2017.
- 6 Davide Ancona, Francesco Dagnino, and Elena Zucca. Modeling infinite behaviour by corules. In *ECOOP’18 - Object-Oriented Programming*, pages 21:1–21:31, 2018.
- 7 Davide Ancona and Agostino Dovier. A theoretical perspective of coinductive logic programming. *Fundamenta Informaticae*, 140(3-4):221–246, 2015.
- 8 Davide Ancona and Elena Zucca. Corecursive Featherweight Java. In *FTfJP’12 - Formal Techniques for Java-like Programs*, pages 3–10. ACM Press, 2012.
- 9 Davide Ancona and Elena Zucca. Safe corecursion in coFJ. In *FTfJP’13 - Formal Techniques for Java-like Programs*, page 2. ACM Press, 2013.
- 10 Pietro Barbieri, Francesco Dagnino, Elena Zucca, and Davide Ancona. Corecursive Featherweight Java revisited. In Alessandra Cherubini, Nicoletta Sabadini, and Simone Tini, editors, *ICTCS’19 - Italian Conf. on Theoretical Computer Science*, volume 2504 of *CEUR Workshop Proceedings*, pages 158–170. CEUR-WS.org, 2019. URL: <http://ceur-ws.org/Vol-2504/paper19.pdf>.
- 11 Henning Basold, Ekaterina Komendantskaya, and Yue Li. Coinduction in uniform: Foundations for corecursive proof search with Horn clauses. In *Programming Languages and Systems - 28th European Symposium on Programming, ESOP 2019, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2019, Prague, Czech Republic, April 6-11, 2019, Proceedings*, pages 783–813, 2019.
- 12 B. Courcelle. Fundamental properties of infinite trees. *Theoretical Computer Science*, 25:95–169, 1983.
- 13 Francesco Dagnino. Coaxioms: flexible coinductive definitions by inference systems. *Logical Methods in Computer Science*, 15(1), 2019. URL: <https://lmcs.episciences.org/5277>.
- 14 E.Komendantskaya et al. A productivity checker for logic programming. *Post-proc. LOPSTR’16*, 2017. arXiv:1608.04415.
- 15 Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler. Featherweight Java: A minimal core calculus for Java and GJ. In *ACM Symp. on Object-Oriented Programming: Systems, Languages and Applications 1999*, pages 132–146. ACM Press, 1999. doi:10.1145/320384.320395.
- 16 Jean-Baptiste Jeannin and Dexter Kozen. Computing with capsules. *Journal of Automata, Languages and Combinatorics*, 17(2-4):185–204, 2012. doi:10.25596/jalc-2012-185.
- 17 Jean-Baptiste Jeannin, Dexter Kozen, and Alexandra Silva. Cocaml: Functional programming with regular coinductive types. *Fundamenta Informaticae*, 150:347–377, 2017.
- 18 E. Komendantskaya et al. Coalgebraic logic programming: from semantics to implementation. *J. Logic and Computation*, 26(2):745, 2016. doi:10.1093/logcom/exu026.

- 19 X. Leroy and H. Grall. Coinductive big-step operational semantics. *Information and Computation*, 207(2):284–304, 2009.
- 20 L. Simon. *Extending logic programming with coinduction*. PhD thesis, University of Texas at Dallas, 2006.
- 21 L. Simon, A. Bansal, A. Mallya, and G. Gupta. Co-logic programming: Extending logic programming with coinduction. In *ICALP 2007*, pages 472–483, 2007.

Perfect Is the Enemy of Good: Best-Effort Program Synthesis

Hila Peleg 

University of California San Diego, CA, USA
hpeleg@eng.ucsd.edu

Nadia Polikarpova 

University of California San Diego, CA, USA
npolikarpova@eng.ucsd.edu

Abstract

Program synthesis promises to help software developers with everyday tasks by generating code snippets automatically from input-output examples and other high-level specifications. The conventional wisdom is that a synthesizer must always satisfy the specification exactly. We conjecture that this all-or-nothing paradigm stands in the way of adopting program synthesis as a developer tool: in practice, the user-written specification often contains errors or is simply too hard for the synthesizer to solve within a reasonable time; in these cases, the user is left with a single over-fitted result or, more often than not, no result at all. In this paper we propose a new program synthesis paradigm we call *best-effort program synthesis*, where the synthesizer returns a ranked list of partially-valid results, i.e. programs that satisfy some part of the specification.

To support this paradigm, we develop *best-effort enumeration*, a new synthesis algorithm that extends a popular program enumeration technique with the ability to accumulate and return multiple partially-valid results with minimal overhead. We implement this algorithm in a tool called BESTER, and evaluate it on 79 synthesis benchmarks from the literature. Contrary to the conventional wisdom, our evaluation shows that BESTER returns useful results even when the specification is flawed or too hard: *i)* for all benchmarks with an error in the specification, the top three BESTER results contain the correct solution, and *ii)* for most hard benchmarks, the top three results contain non-trivial *fragments* of the correct solution. We also performed an exploratory user study, which confirms our intuition that partially-valid results are useful: the study shows that programmers use the output of the synthesizer for comprehension and often incorporate it into their solutions.

2012 ACM Subject Classification Theory of computation → Program specifications; Software and its engineering → Automatic programming

Keywords and phrases Program Synthesis, Programming by Example

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2020.2

Supplementary Material ECOOP 2020 Artifact Evaluation approved artifact available at <https://doi.org/10.4230/DARTS.6.2.16>.

Funding This work has been supported by the National Science Foundation under Grant 1911149.

1 Introduction

Program synthesis has emerged as a promising technology for automating low-level programming tasks [24, 50, 54, 3]. For software developers, program synthesis can be an attractive alternative to online help forums when it comes to “opportunistic programming” [11], or hunting for code that will perform a small subtask needed in a larger development task. Using a *Programming by Example* (PBE) synthesizer [36, 21, 20, 19, 46, 25, 56], developers can specify the desired behavior with a set of input-output examples (or unit tests), and the synthesizer would generate a code snippet that satisfies each of the examples.



© Hila Peleg and Nadia Polikarpova;

licensed under Creative Commons License CC-BY

34th European Conference on Object-Oriented Programming (ECOOP 2020).

Editors: Robert Hirschfeld and Tobias Pape; Article No. 2; pp. 2:1–2:30

Leibniz International Proceedings in Informatics



LIPICs Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany



Although PBE techniques have made great strides in recent years and have been used successfully in end-user tools [23, 31, 29], they have not seen wide adoption in mainstream software development. We conjecture that one important reason is that existing synthesizers follow an “all-or-nothing” paradigm: they either return a program that is correct on *all examples*, or fail. In practice, however, humans make mistakes, so examples might contain errors. Even if all the examples are correct, the program might just be too complex for the synthesizer to generate: no matter how much we improve the synthesizer, there will always be problems it fails to solve within the amount of time that the user is willing to wait. In these cases, all-or-nothing synthesis is utterly useless to the programmer: it either returns a single over-fitted result (that satisfies the erroneous specification) or, more often than not, no result at all. Iterative synthesizers [32, 39, 7] offer a partial remedy by allowing the user to refine a problematic specification, but they still waste user’s time in the unsuccessful iterations.

We believe that turning PBE synthesizers into useful mainstream programming tools requires addressing two core *challenges*:

- 1) **Erroneous specifications:** How can we make the synthesizer robust to small errors in the specification?
- 2) **Hard problems:** How can we make the synthesizer helpful even if it cannot solve a problem completely?

Switching paradigms

To address the two core challenges, we need to abandon the all-or-nothing view of synthesis and instead take the approach of successful code completion tools: an imperfect result is better than no result, as long as it is indicated as such. To this end, we propose a new PBE paradigm we dub *best-effort program synthesis*, in which the user provides examples, and the synthesizer returns a shortlist of partially-valid results, i.e. programs that satisfy at least some of the examples. Previous work has shown that *a*) partially-valid programs often share non-trivial fragments with the correct solution [46], and *b*) users prefer editing incorrect code to writing code from scratch [13]. Hence it is reasonable to assume that partially-valid results help the user move forward both when the specification contains errors (by generating a solution for the error-free subset of the examples) and when the problem is too hard (by generating a special-case program that can be used as a building block in the final solution).

Efficient best-effort synthesis

A naive way to implement best-effort synthesis would be to use an existing synthesizer as a black box and re-run it again and again with different subsets of the specification, displaying any generated programs to the user. This is highly inefficient, however, especially when the original synthesis problem takes too long to solve: in this case, some specification subsets may still take too long. Ideally, we would like to deliver partially-valid results without requiring the synthesizer to do more work.

Our core *technical insight* is that a popular program search algorithm – bottom-up enumeration with observational equivalence reduction [55, 2] – can be extended to accumulate partially-valid results during search with minimal overhead. The extension is possible because this search algorithm is *monotonic* in the set of examples: the set of programs explored with the full specification includes all programs that would be explored with a partial specification. We formalize this monotonicity property and our extended *best-effort enumeration* algorithm in Section 3.

Ranking partially-valid results

In general, there can be too many partially-valid results to display them all to the user, so a best-effort synthesizer needs a way to automatically select a manageable number of results (3–5) that are most likely to be useful to the programmer. It is common in program synthesis to introduce a *ranking function* for the generated programs and present top $N \geq 1$ ranked results to the user [23, 28, 43]. For the best-effort setting, we design a ranking function that incorporates both *syntactic* and *semantic* features of programs, such as simplicity and the number of examples satisfied. The details of the ranking are described in Section 4.

Evaluating best-effort solutions

We implement our approach in a tool called BESTER (Best-Effort Synthesis TERminal), which gives users access to a best-effort synthesizer from a Read-Evaluate-Print-Loop (REPL). We evaluate BESTER on 79 benchmarks we collected from the 2017 SYGUS competition [4] and the EUPHONY benchmark suite [33]. Our evaluation shows that *i*) BESTER can overcome errors in the specification and still return the correct solution in the top three results, *ii*) when a synthesis problem is hard and times out, BESTER still returns useful fragments of the solution, and *iii*) BESTER’s ability to solve correct specifications is not impacted (Section 5). Moreover, BESTER compares favorably to the naive approach of using a state-of-the-art synthesizer¹ as a black box and eliminating examples from the specification one by one.

We also performed a small exploratory user study of BESTER, in which programmers used BESTER to solve tasks in an unfamiliar programming language; the tasks were too hard for the synthesizer to solve completely within 40 seconds (Section 6). Our study shows that programmers make use of synthesis results for comprehension, both of the task and of the language, and that programmers often incorporate synthesis results into their solutions either by copy-pasting or by editing a partially-valid solution until it fully satisfies the examples.

Main contributions

To summarize, this paper makes the following contributions:

1. *Best-effort program synthesis*: a new user interaction paradigm for PBE that is likely to yield helpful results even when the problem is ill-specified or too hard to solve completely.
2. *Best-effort enumeration*: an algorithm for efficiently collecting partially-valid solutions during enumerative synthesis.
3. A ranking function for partially-valid solutions that incorporates both syntactic and semantic properties of programs, and performs well in our experiments.
4. BESTER: a prototype implementation of best-effort synthesis, shown both empirically and in an exploratory user study to be robust to specification errors and to produce useful program fragments on hard problems.

2 Overview

In this section, we consider a scenario that requires best-effort synthesis.

¹ We used CVC4 [44], the winner of the 2017–2019 SYGUS competitions in the PBE-Strings category.

```

> (- (str.len arg0) (str.len (str.replace arg0 "\n" "")))
+-----+-----+-----+
| input          | result | expected |
+-----+-----+-----+
| arg0 -> "one"  | 0      | 0         |
+-----+-----+-----+
| arg0 -> "one\ntwo" | 1      | 1         |
+-----+-----+-----+
| arg0 -> "one\ntwo\nthree" | 1      | 2         |
+-----+-----+-----+
| arg0 -> "one\ntwo\nthree\nfour" | 1      | 3         |
+-----+-----+-----+
> |

> :s
Synthesizing... (Press any key to interrupt)
Current best: [3/4]
1: (- (str.len arg0) (str.len (str.replace (str.replace arg0 "\n" "") "\n" ""))) [3/4]
2: (str.len (int.to.str (str.len arg0))) [2/4]
3: (str.indexof arg0 (str.at arg0 -1) (str.indexof arg0 "\n" 1)) [2/4]
4: (str.indexof "" (str.at arg0 -1) (str.indexof arg0 "\n" 1)) [2/4]
5: (str.indexof "\n" (str.at arg0 -1) (str.indexof arg0 "\n" 1)) [2/4]
> :1
+-----+-----+-----+
| input          | result | expected |
+-----+-----+-----+
| arg0 -> "one"  | 0      | 0         |
+-----+-----+-----+
| arg0 -> "one\ntwo" | 1      | 1         |
+-----+-----+-----+
| arg0 -> "one\ntwo\nthree" | 2      | 2         |
+-----+-----+-----+
| arg0 -> "one\ntwo\nthree\nfour" | 2      | 3         |
+-----+-----+-----+

```

(a) Evaluating user-written program on the examples. (b) Best-effort synthesis results.

■ **Figure 1** The BESTER REPL interface.

2.1 A motivating example

Our example is derived from one of the benchmarks in the PBE-Strings track of the SYGUS (Syntax-Guided Synthesis) competition [5, 4]. In this competition, synthesizers are expected to generate programs in a simple language of S-expressions with built-in operations on integers (such as `+` or `-`) and strings (such as `str.len` and `str.replace`). A benchmark in the PBE-Strings track is given by a set of input-output examples and a grammar that defines the space of candidate programs (i.e. the relevant subset of the SYGUS language). These benchmarks mimic small tasks performed by programmers, and some are directly derived from StackOverflow questions.

In this scenario, a programmer is attempting to solve a task that asks them to count the number of line breaks in a string. They are using a development environment enriched with a synthesizer: they have the option to invoke the synthesizer at any point during development and incorporate (fragments of) its output into their own code.

The programmer starts by providing a set of test cases (examples):

```

e0 = "one" → 0
e1 = "one\ntwo" → 1
e2 = "one\ntwo\nthree" → 2
e3 = "one\ntwo\nthree\nfour" → 3

```

We notice, though the user does not, that e_3 contains a typo in the string and would, given the expected program, only return 2 rather than 3.

The user then attempts to write a program to satisfy their test cases by computing the difference in length between the input string, `arg0`, and `arg0` with newlines removed:

```
(- (str.len arg0) (str.len (str.replace arg0 "\n" "")))
```

The user executes their tests, and only e_0 and e_1 pass, as shown in Figure 1a. They might not immediately realize that the reason for this behavior is the unexpected semantics of `str.replace` in the SYGUS language, which only replaces the first instance of the substring rather than all instances. Because e_2 fails as well as e_3 , the typo in e_3 goes unnoticed.

At this point, the user decides to delegate solving the task to the synthesizer. Running the state-of-the-art synthesizer CVC4 [44] on this synthesis query yields the result:

```

(ite (str.contains (str.replace arg0 "\n" "") "\n")
  (ite (str.suffixof (str.at arg0 (str.len "\n")) arg0)
    (str.len "\n") (str.indexof arg0 "\n" 1))
  (ite (str.prefixof arg0 (str.replace arg0 "\n" arg0)) 0 1))

```

This program satisfies all the test cases provided to the synthesizer, but it is so complex that the user will most likely discard it without reading and be none the wiser about the typo in the tests or their misconception about the semantics of `str.replace`.

Running our tool BESTER, on the other hand, produces a ranked list of synthesis results, as shown in Figure 1b. The first result in this list is:

```
(- (str.len arg0) (str.len (str.replace (str.replace arg0 "\n" "") "\n" "")))
```

which is relatively simple and in fact similar to the user's initial solution (except that it calls `str.replace` on the input string twice). Contrasting the outputs of the initial program and this result helps the user realize their misconception about `str.replace`, while the tool's failure to solve e_3 is likely to call their attention to the typo.

Best-effort synthesis for hard specifications

Consider a slightly different specification our programmer could have provided, where examples e_0, e_1, e_2 are as before, but example e_3 is replaced with

$$e'_3 = \text{"one\ntwo\nthree\nfour\nfive\nsix\nseven\neight"} \rightarrow 7$$

The programmer asks the (traditional) synthesizer for help, but after 30 seconds of waiting, their patience is exhausted, and they interrupt the synthesizer before it can produce any results. The reason this problem is taking so long to solve is that the SyGuS language contains no general solution that works for an arbitrary number of newlines, so the shortest program that satisfies e'_3 contains seven calls to `str.replace`; programs of this size present a challenge for state-of-the-art synthesizers. Once again, the user just wasted their time and is back to square one.

Although this particular example seems contrived, the general scenario where the user is unaware of the limitations of the synthesis algorithm and gives it more than it can handle, is very common. If the programmer is using BESTER, however, and interrupts it after 30 seconds, they would get exactly the same set of results as in the previous scenario, shown in Figure 1b. This is because BESTER always searches for solutions to all subsets of input examples simultaneously, and the solution for $\{e_0, e_1, e_2\}$ is much smaller – and hence will be discovered much earlier – than the solution for the full set of examples.

2.2 Background: Observational Equivalence Reduction

Before we explain how BESTER is able to generate such partially-valid results efficiently, we must introduce the baseline synthesis technique we build upon: bottom-up enumeration with *observational equivalence reduction* [55, 2], or OE-reduction for short. Program synthesizers work by searching a space of candidate programs until they encounter one that satisfies the specification. The central challenge of program synthesis is the astronomically large size of the search space, so different synthesis techniques find different ways to reduce the space, i.e. exclude large chunks of the space from consideration.

For illustration purposes, in this section we will consider the program space defined by an artificially small grammar, shown in Figure 2a. This grammar allows using only two integer literals (0 and 3), one string literal (" "), a single variable (`input`), and three operations: `+`, `str.indexof`, and `str.substr`.

Bottom-up enumeration

Bottom-up enumeration is a synthesis technique that maintains a *bank* of enumerated programs and constructs new programs by applying production rules to programs from the bank. Recall the grammar in Figure 2a. We begin enumeration with an empty bank, so in the first iteration we are limited to production rules that require no subexpressions – literals and variables; this yields the programs `0`, `3`, `" "`, and `input`, which are added to the bank. In the following iterations, production rules that require subexpressions are applied to the programs in the bank: for example, the rule $Int \rightarrow (+ Int Int)$ is applied to all pairs of *Int* expressions, creating new programs `(+ 0 0)`, `(+ 0 3)`, `(+ 3 0)`, and `(+ 3 3)`, as seen in Figure 2b.

The enumeration is generally performed in the order of height: we first construct all programs of height 0, then height 1 and so on; each iteration constructs all programs of height $n + 1$ using the programs of heights up to n stored in the bank. As a consequence, discarding even a few programs from the bank can drastically reduce the number of programs to be enumerated in future iterations.

Equivalence reduction

A natural candidate for discarding from the bank is a redundant program, i.e., a program that is functionally equivalent to another program in the bank. In our example, the program `(+ 0 3)` is functionally equivalent to the program `3`, and hence can be safely discarded. State-of-the-art bottom-up synthesizers [55, 2, 6] use a more aggressive notion of program equivalence called *observational equivalence*, which is also easier to check: two programs are considered equivalent if they evaluate to the same output for every input in the user-provided set of examples.

► **Example 1.** Let us assume two pairs of input-output examples

```
e0 = "The Demolished Man" → "Demolished"
e1 = "The Stars My Destination" → "Stars"
```

We follow the enumeration of programs with OE-reduction, summarized in Figure 2b.

First, we create an *input vector*, which in this case contains two inputs:

```
⟨"The Demolished Man", "The Stars My Destination"⟩
```

The algorithm evaluates each constructed program point-wise on the input vector, producing an *output vector*. Two programs are deemed observationally equivalent if their output vectors are equal.

Height 0: First we enumerate programs of height 0 (programs 1–4 in Figure 2b). The program `0` is a literal and evaluates to 0 on every input, resulting in the output vector $\langle 0, 0 \rangle$. Likewise the programs `3` and `" "` result in $\langle 3, 3 \rangle$ and $\langle " ", " " \rangle$ respectively. The program `input` (the input variable) yields the output vector $\langle \text{"The Demolished Man"}, \text{"The Stars My Destination"} \rangle$. Since all four output vectors are different, all four programs are added to the bank.

Height 1: Next, we enumerate programs of height $n + 1$ by applying production rules in the grammar to programs from the bank at heights up to n (in this case, up to 0). The production rule for `str.indexof` requires two arguments of type string, and will be applied to

```

Start → String
Int  → 0 | 3
      | (+ Int Int)
      | (str.indexof String String)
String → " " | input
        | (str.substr String Int Int)

```

(a) A small grammar in the SYGUS format. Notice that the language is limited to the literal constants that appear here.

#	program	output on e_0	output on e_1	equivalent to
1	0	0	0	
2	3	3	3	
3	" "	" "	" "	
4	input	"The Demolished Man"	"The Stars My Destination"	
5	(+ 0 0)	0	0	#1
6	(+ 0 3)	3	3	#2
7	(+ 3 0)	3	3	#2
8	(+ 3 3)	6	6	
9	(str.indexof " " " ")	0	0	#1
10	(str.indexof " " input)	-1	-1	
11	(str.indexof input input)	0	0	#1
12	(str.indexof input " ")	3	3	#2

(b) An enumeration of the grammar by height.

■ **Figure 2** The enumeration in Example 1. Programs are generated from the grammar by height, first productions requiring only a terminal, and next productions requiring a subtree, taken from previously seen programs.

all combinations of string programs of height 0. This will produce, among others, the program `(str.indexof " " " ")` with the output vector $\langle 0, 0 \rangle$. Notice that the bank already contains a program with this vector: the program 0. The algorithm therefore discards `(str.indexof " " " ")` and does not add it to the bank. In general, the algorithm maintains an *invariant* that the bank contains at most one representative of any observational equivalence class.

The same production rule also generates the program `(str.indexof input " ")`. This program seems helpful for solving the given examples; however, its output vector is $\langle 3, 3 \rangle$, whose equivalence class already has a representative, the program 3, so the program `(str.indexof input " ")` will be discarded. Unlike in the case of `(str.indexof " " " ")`, this seems an imprudent decision. However, it is in fact sound to do so *for these inputs*: so long as we do not care about differently structured inputs, `(str.indexof input " ")` and 3 are completely interchangeable. If the user introduces another example with a new input such as "Virtual Unrealities", the new extended output vectors will be $\langle 3, 3, 3 \rangle$ and $\langle 3, 3, 7 \rangle$, and the two programs will no longer be equivalent.

2.3 Our approach

Next we describe how BESTER modifies the baseline OE-reduction enumeration technique from the previous subsection to maintain a ranked list of partially-valid programs. If the search happens to encounter a program that fully satisfies the specification, it stops; otherwise, if

the search is interrupted before a solution was found, BESTER simply returns the current list of partially-valid results to the user. We refer to this modification of OE-reduction search as *best-effort enumeration*; Section 3 details the search algorithm and its correctness.

Searching for all example subsets

Recall the task from Section 2.1, where the user is trying to count line breaks in a string, but has an error in the example e_3 . We would like to show the programmer the following partially-valid yet useful program p^* , which satisfies examples $\{e_0, e_1, e_2\}$:

```
(- (str.len arg0) (str.len (str.replace (str.replace arg0 "\n" "") "\n" "")))
```

Since we do not know a-priori which subset of examples would yield a useful result, we would like the synthesizer to simultaneously search for programs satisfying *all* non-empty subsets of $\{e_0, e_1, e_2, e_3\}$ (thus, including $\{e_0, e_1, e_2\}$).

Note that many synthesis techniques are not amenable to such simultaneous search: for example, in constraint-based synthesis [52, 26], a run of the synthesizer with the full set of examples would never construct p^* , because it does not satisfy e_3 . We observe that unlike most synthesis techniques, the OE-reduction algorithm has the ability to maintain solutions for all example subsets with little to no overhead, thanks to a curious monotonicity property: *adding a new example never excludes programs from the enumeration*.

Let us illustrate this property on our running example. Consider a *hypothetical* run of an OE-synthesizer on the examples $\{e_0, e_1, e_2\}$, and assume that in this run p^* is added to the bank. We conclude that p^* is the first program the synthesizer constructed that produces the output vector $\langle 0, 1, 2 \rangle$, and hence has been chosen as the representative of the $\langle 0, 1, 2 \rangle$ equivalence class. Now consider the *actual* run of the synthesizer, on the full set of examples $\{e_0, e_1, e_2, e_3\}$; we argue that in this run p^* must be chosen as the representative of the $\langle 0, 1, 2, 2 \rangle$ equivalence class and cannot be discarded by OE reduction. To see why, assume a different program p' is chosen as the representative; then p' would have been enumerated before p^* and would also return $\langle 0, 1, 2 \rangle$ on the first three examples; but this contradicts our assumption that p^* is the representative for $\langle 0, 1, 2 \rangle$.

In other words, since each additional example *refines* the partition of the program space, the bank in the actual run must be a superset of the bank in the hypothetical run. Moreover, the output vector of each program in the bank is already computed as part of performing OE-reduction, and compared to the expected output vector; hence, performing a slightly more complex check for the purpose of identifying partially-valid results incurs only minimal overhead.

Ranking best-effort candidates

A best-effort enumeration as described above might accumulate multiple results satisfying each subset of the examples. However, we cannot simply show them to the user in the order in which they are discovered: trivial programs such as a literal or variable satisfying one or two of the examples would be discovered immediately, but would often be a poor candidate. For instance, in the example from Section 2.1, the program 0 satisfies $\{e_0\}$, the program `(str.indexof arg0 "\n")` satisfies $\{e_4\}$ (the erroneous example), and the program `(ite (str.contains arg0 "\n") 1 0)` satisfies $\{e_0, e_1\}$. All of these will be discovered fairly early on in the enumeration.

Instead, the partially-valid programs in the bank need to be *ranked* so that a manageable number (no more than 5) of promising programs can be returned to the user. We have developed a simple ranking function for BESTER that takes into account both syntactic and

semantic properties of programs, and performs well empirically. Section 4 details our ranking function and discusses other possible rankings. Intuitively, our ranking rewards programs that satisfy more examples, programs that use all of their inputs (the so called *relevancy requirement* inspired by other synthesis techniques [20, 27]), smaller programs, and programs where the incorrect outputs are close to the expected outputs. Among the programs listed above, `(str.indexof arg0 "\n")` and `0` both satisfy one example, but the former is preferred by our ranking because it uses its input.

3 Best-Effort Enumeration With Observational Equivalence

In this section, we detail the way an enumerative search with observational equivalence can be used to find and rank best-effort results to a synthesis query.

Let us consider the challenge in finding a best-effort solution. Since the set of user-provided examples \mathcal{E} might be unsatisfiable, we wish to return a program that satisfies some $\mathcal{E}^* \subseteq \mathcal{E}$. However, we do not know in advance whether \mathcal{E} is satisfiable, and if it is not, *which* \mathcal{E}^* we are searching for a solution to.

We can address this challenge with minimal effort thanks to several properties of equivalence classes.

Refined equivalence classes

Enumerative synthesis with observational equivalence adds only one representative from each equivalence class to its bank of programs based on an equivalence relation \equiv_I defined as follows:

$$p_1 \equiv_I p_2 \iff \forall \iota \in I. \llbracket p_1 \rrbracket(\iota) = \llbracket p_2 \rrbracket(\iota)$$

where the equality of execution results considers outputs, exceptions, and side effects. In a PBE synthesis query, the inputs in I are derived from the example set \mathcal{E} such that $I = \{\iota \mid (\iota, \omega) \in \mathcal{E}\}$.

If the enumeration that has already added to the reduced program bank the program p encounters a program p' such that $p \equiv_I p'$, a decision is made which one will be the *representative of the equivalence class* $[p]$ that both p and p' inhabit. The representative is then kept in the program bank and the other program is discarded. In most synthesizers that perform the enumeration in layers (i.e., first programs of height 0, then of height 1, etc.), the first program encountered from each equivalence class is selected as its representative, as was shown in Figure 2b.

Now consider $\mathcal{E}' \subset \mathcal{E}$, a non-empty subset of examples, and its input set I' . It is easy to see that \equiv_I is a *refinement* of $\equiv_{I'}$, since it is the intersection of $\equiv_{I'}$ and $\equiv_{I \setminus I'}$. This means that \equiv_I refines the partition into equivalence classes made by $\equiv_{I'}$, or that for a program p in the candidate program space, $[p]_{\equiv_I} \subseteq [p]_{\equiv_{I'}}$.

We notice that if selection of the representative is deterministic, then if p was the representative of $[p]_{\equiv_{I'}}$, the less refined (and possibly larger) equivalence class, then p will also be the representative of $[p]_{\equiv_I}$: representative selection has determined p to be the representative against each of the programs in $[p]_{\equiv_I}$ when it was decided to be the representative of $[p]_{\equiv_{I'}}$.

This means that if p was included in the bank of programs in a less refined enumeration with OE-reduction, p will be in the program bank of a more refined enumeration, one with more examples.

■ **Algorithm 1** A best-effort enumeration.

```

Input:  $\mathcal{E}$  a user-provided example specification,  $\mathcal{G}$  a grammar,  $f$  a fitness function,
          $maxResults$  the maximum number of results to return to the user
Result: Top  $maxResults$  synthesized programs
1   $programBank \leftarrow \emptyset$ 
2   $resultCandidates \leftarrow PriorityQueue()$ 
3  while timeout has not passed do
4  |   foreach  $prodRule \in \mathcal{G}$  do
5  | |    $k \leftarrow arity(prodRule)$ 
6  | |   foreach  $(arg_1, \dots, arg_k) \in programBank^k$  do
7  | | |   if  $(arg_1, \dots, arg_k)$  is suitable for prodRule then
8  | | | |    $newProg \leftarrow prodRule(arg_1, \dots, arg_k)$ 
9  | | | |   if  $\forall p \in programBank. p \not\equiv_I newProg$  then
10 | | | | |   /* Found the representative of a new equivalence class,
11 | | | | |   add to the bank */
12 | | | | |    $programBank \leftarrow programBank \cup \{newProg\}$ 
13 | | | | |    $exec \leftarrow \{(\iota, \llbracket newProg \rrbracket(\iota)) \mid \iota \in I\}$ 
14 | | | | |   if  $exec \cap \mathcal{E} \neq \emptyset$  then /* newProg partially satisfies  $\mathcal{E}$  */
15 | | | | | |    $resultCandidates.insertWithPriority(newProg, f(newProg, \mathcal{E}))$ 
16 | | | | |   end
17 | | | |   end
18 | |   end
19 |   end
20 end
21 end
   /* Either timeout has passed and or a fully satisfying program was
   found. We now return a list of options by rank. */
22  $results \leftarrow List()$ 
23 for  $i = 1$  to  $\min(maxResults, resultCandidates.size())$  do
24 |    $results.append(resultCandidates.getFront())$ 
25 end
26 return  $results$ 

```

Notice that, despite the use of an inputs *vector* in Section 2.2 (and in practical implementations), the operations are unordered. This means that it does not matter which of the examples are missing from \mathcal{E}^- for the property to hold.

3.1 Finding best-effort solutions

Fortunately, since performing observational equivalence with \mathcal{E} is a refinement of any strict, nonempty subset of \mathcal{E} , we can essentially test all nonempty subsets of \mathcal{E} simultaneously. Representative selection ensures we will see all programs we would see enumerating a subset of the examples, so we can simply collect programs that satisfy any of the examples, instead of ones that satisfy *all* of them.

Lines 4 – 8 of Algorithm 1 are a simple bottom-up enumeration of the space, applying each of the production rules to each of the programs previously added to the program bank, generating additional programs. Lines 9 – 10 are the implementation of the OE-reduction, adding to the program bank only programs that are the first of their equivalence class to be encountered. Line 15 is the stopping condition for any PBE synthesizer: whether executing each input leads to its expected output. It is simply lines 12 – 14 that “piggyback” on the enumeration with observational equivalence, collecting programs that satisfy any of the examples and create the best-effort search.

This means that when enumerating the example in Section 2.1, the program

```
(- (str.len arg0) (str.len (str.replace (str.replace arg0 "\n" "") "\n" "")))
```

is produced by the algorithm on line 8. In a regular observational equivalence reduction, the program will be added to the reduced program bank on line 10 for use in enumerating larger programs, and the next step would be to perform the check on line 15, testing whether it fully satisfies the specification. Since it satisfies 3 of the 4 examples, a simple enumeration would not return it and enumeration would continue searching for a single fully-satisfying program to show the user.

In a best-effort enumeration, the condition on line 12 admits programs that satisfy any nonempty subset of \mathcal{E} . The program is added to the list of best-effort results, of which the best results will be returned to the user.

The correctness proposition of observational equivalence [2] guarantees that if a program that satisfies \mathcal{E} exists in the space, we will encounter exactly one such program, as other programs satisfying \mathcal{E} are in its equivalence class and are not part of the reduced program space. However, if we consider any strict subset, this guarantee no longer holds: when partitioning the space of programs possible in the grammar based on observational equivalence for \mathcal{E} , any $\mathcal{E}' \subset \mathcal{E}$ is now represented by a number of equivalence classes in the program space instead of just one. In other words, more than one program satisfying \mathcal{E}' may be encountered in the course of the enumeration.

This means there are two dimensions in which our goal is no longer unique: along an enumeration, we are looking for a program that satisfies one of exponentially many $\mathcal{E}' \subseteq \mathcal{E}$, and there can be many such programs for each \mathcal{E}' . However, since the results of a best-effort enumeration are intended for consumption by a user, we must limit ourselves to a small number of returned results. This means that in the course of an enumeration based on \mathcal{E} programs that satisfy any nonempty subset of \mathcal{E} are collected, and the *best* few are returned to the user. This is determined by a fitness function used to rank the programs in line 13 of Algorithm 1.

We will introduce our fitness function in the next section.

4 Fitness Function

As we have shown in Sections 2 and 3, more than one program can satisfy the same number of specifications. In this section, we discuss the considerations in constructing the fitness function used in our implementation of BESTER, and suggest additional parameters that could be added for other synthesizers.

The composition of the function is:

$$f(p, \mathcal{E}) = 3 \cdot \text{satisfied}(p, \mathcal{E}) + 2 \cdot \text{relevancy}(p) + \text{distance}(p, \mathcal{E}) + \text{size}(p)$$

We now break down each of these elements.

Examples satisfied

Since a program satisfying one example and a program satisfying all examples but one are not equally good, we use the portion of examples satisfied in our ranking of the program.

$$satisfied(p, \mathcal{E}) = \frac{|\{(\iota, \omega) \in \mathcal{E} \mid \llbracket p \rrbracket(\iota) = \omega\}|}{|\mathcal{E}|}$$

This portion of the fitness function is the most strongly weighted, as we still give the most importance to the best effort, i.e. solving the largest portion of the specification.

Relevancy

Given two programs that solve the same number of examples, we prefer one that uses more of its input. For example, let us assume a grammar with two input variables, `arg0` and `arg1`, and three programs that satisfy 2 of 3 examples in \mathcal{E} :

```
p1 = true
p2 = (str.contains arg0 " ")
p3 = (str.prefixof arg1 arg0)
```

Intuitively, we are certain we want $f(p_1)$ to be the lowest of the three, but in all likelihood, we also want to reward p_3 for using all available input from the user. This is a tactic employed by other synthesis tools such as [20, 27].

We define for all variables \mathcal{V} available in the grammar:

$$relevancy(p) = \frac{|\{var \in \mathcal{V} \mid var \in p\}|}{|\mathcal{V}|}$$

Distance from output

While we strongly reward a program for each satisfied example, we also wish to reward programs that do “better” with regard to the remaining examples.

Currently we include this element only for synthesis tasks that search for a string program. For strings, being closer to the expected output can be seen as returning a subset or superset of it, or constructing a close string. This is easily rewarded by using Levenshtein Distance [34] to measure the distance of the *unsatisfied* example results from the intended output. While this component may not be suitable for numeric types, for other structured types such as lists or trees, other such structured distance metrics can be employed in place of *LD*.

We denote $\mathcal{E}^- = \{(\iota, \omega) \in \mathcal{E} \mid \llbracket p \rrbracket(\iota) \neq \omega\}$ to be the unsatisfied examples, and define:

$$distance(p, \mathcal{E}) = \begin{cases} \text{avg}_{(\iota, \omega) \in \mathcal{E}^-} \left(\left\{ 1 - \frac{LD(\llbracket p \rrbracket(\iota), \omega)}{\max(|\omega|, |\llbracket p \rrbracket(\iota)|)} \right\} \right) & p \text{ is a string program and } |\mathcal{E}^-| > 0 \\ 0 & \text{o.w.} \end{cases}$$

While we include this in the fitness function, we do not weight it as high as some of the other components as we do still want to allow other logic that may help the user toward the correct answer, e.g., constructing a complement of the result in order to remove it, to rank well and be displayed.

Program size

Finally, we incorporate the size of the program into the function. In a regular enumerative synthesizer, ranking by size is implicit, as programs of a lower height will be reached first. Since programs of a lower height are *simpler programs*, this tactic is employed in many synthesizers. In best-effort synthesis we may encounter programs of very different sizes that satisfy the same examples before we reach the timeout. We therefore add the height of the program into the ranking to prefer shorter ASTs.

Additionally, we would like to distinguish between programs of the same height. To do this, we use $terms(p)$, the number of nodes in the AST of p . For example, $p_1 = (\text{str.at arg0 (+ 1 1)})$ and $p_2 = (\text{str.++ (str.++ " " " ") (str.substr arg0 1 1)})$ are both programs of height 2, but $terms(p_1) = 5$ whereas $terms(p_2) = 8$.

Since programs are eventually displayed to a user, given two programs of the same height that are indistinguishable by other parameters, we would like to show the user first the one that is easier to read, or the overall-smaller one.

Together, we define:

$$size(p) = \frac{1}{height(p) + 1} + \frac{1}{terms(p)}$$

Including other data

In a domain where not all specifications are created equal, some may be ranked as more important than others. For instance, examples that detail an error scenario may be deemed more or less important than examples that specify a simple output value. Likewise, if not all specifications are examples [40], an importance ranking between different specification types can be used to decide which are more likely to be dropped.

Finally, we address the fact that our fitness function is not learned. In theory, a model could be trained to compute a fitness function according to desired program rankings, or to provide features for a fitness function (e.g., [8, 33] compute the probability of a program, which in their tool is used to speed up the search but could also be used for simple numerical ranking). However, the pool of programs is small, and creating a dataset of ranked best-effort programs large enough to train from, either manually or automatically, would be unreliable at best. In addition, our fitness function, both in selected features and in their weights, encodes in it what we consider to be the important aspects of a best-effort program, rather than numbers overfitted to a small dataset.

5 Empirical Evaluation

In this section we detail the empirical evaluation performed to validate our approach. Our experiments are based on the benchmarks of the SYGUS competition [4] and EUPHONY [33].

Implementation

We implemented an enumerating, observational equivalence synthesizer for the SYGUS language in Scala, then augmented it for best-effort enumeration². Best-effort solutions are accumulated as the enumeration progresses, and the top 5 results are returned. The enumeration loop of our synthesizer has a 40s timeout, selected since it is a manageable length of task interruption for a human user [37].

² <https://github.com/peleghila/bester>

Benchmarks

We used a set of 79 synthesis queries from the 2017 SYGUS competition and the EUPHONY benchmarks. These benchmarks contain a selection of data wrangling and string transformation tasks: the SYGUS benchmarks are entirely string to string transformations but 19 of the EUPHONY benchmarks either have a non-string parameter or synthesize a numeric or boolean expression. Duplicate tasks between SYGUS and EUPHONY were removed from the original benchmark set, as well as benchmarks requiring recursion.

We initially divided them into two sets using a simple OE-based enumerating synthesizer (that does not collect best-effort results): 63 that can be solved within 40s, denoted “easy”, and 16 that cannot, denoted “hard”.

We then created a modified version of the benchmarks in the “easy” set by adding erroneous examples such as typos, off-by-one errors, etc. This was done manually and required great care in order to make sure that the additions are *i*) not consistent with the original target program, and *ii*) do not always create a new example set that is easily generalized. Of 37 modified benchmarks, two contain more than one erroneous example.

We note that while we introduced errors, it is near impossible to introduce *contradictions*, short of pairing the same input with two different outputs. Since most of the SYGUS and EUPHONY benchmarks include the conditional `ite` in their grammar, given enough time the inconsistency in the examples in many of the modified benchmarks can be overcome with case-splitting. The exception to this is a result that requires string constants not included in the grammar and that cannot be generated from the input.

For convenience, we use the simple OE synthesizer to make a distinction between the modified benchmarks:

1. “no-solution”: benchmarks in which the synthesizer does not find a program that satisfies all examples within the 40s timeout, and
2. “overfitted”: benchmarks in which the synthesizer is able to find a solution to the given examples (this solution will usually be long and overfitted via multiple case splits).

Since the origin of many of our benchmarks is the `PBE-Strings` track of the SYGUS competition, we take as state-of-the-art the synthesizer/solver CVC4 [44], winner of the `PBE-Strings` track of the competition since 2017. We use CVC4 1.7, the most recent version available.

Experimental setup

We generated gold-standard solutions for each of the original, unmodified 79 benchmarks. Our gold standard is more forgiving than the SYGUS competition, including both hand-written solutions for the task in the benchmark, as understood by the authors, and solutions from CVC4 that cover all examples, despite taking a different approach. Solutions by CVC4 were accepted as-is, in order to use it as a baseline, despite the fact that, as seen in Section 2.1, those solutions are at times overfitted and full of case-splits, but for every such case a hand-crafted gold-standard solution was also added.

All benchmarks were run on a Lenovo laptop with a i7 quad-core CPU @ 2.60GHz with 16GB of RAM.

Research questions

RQ1: Can Bester discard contradicting examples better than a naive search using a state-of-the-art tool? To test this, we examine the result of running BESTER on the “no-solution” portion of the modified benchmark set. We run BESTER with a 40s timeout, which is not enough for a simple enumerating synthesizer to find a

satisfying program for these tasks. We then test whether a gold-standard program for the original benchmark was returned as the top-ranked result, and compare to the ability of CVC4 to find the gold-standard result when run first with the full example set and then with reduced example sets.

- RQ2: Can Bester rank a gold-standard result high when there is an overfitted, untended result for the example set?** To test this, we examine the “overfitted” portion of the modified benchmark set. We still ran BESTER with a 40s timeout, but since a fully satisfying result exists, these benchmarks terminate before the timeout. Though BESTER will find a fully-satisfying result to the examples, it will also return other best-effort results. We search for a gold-standard solution in the top 3 results for each task.
- RQ3: Can Bester find pieces of a gold-standard solution when the task is too hard for it to synthesize?** To test this, we search for pieces of gold-standard solutions in the top results when enumerating the “hard” benchmark set. This question is further examined in the user study in Section 6.
- RQ4: Does the best-effort enumeration in Bester interfere with its ability to solve a simple synthesis task?** In other words, can BESTER solve the “easy” benchmark set, returning the gold-standard solution as the top-rated result?

5.1 Erroneous examples

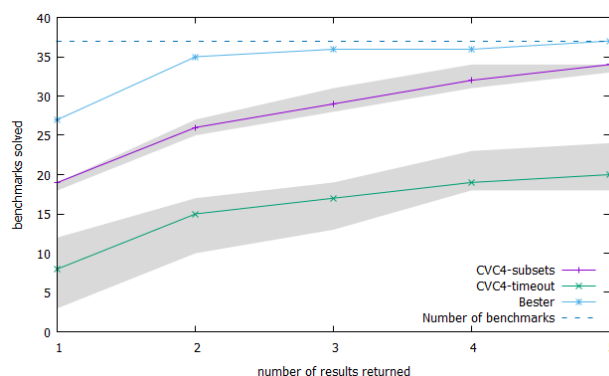
In RQ1 and RQ2, we wish to empirically quantify the effort of a user looking at a list of results. That the gold-standard solution appear *somewhere* on the list of programs shown as a result to a synthesis call is necessary but insufficient. Ideally, the user would have to look through as few programs as possible until they find the one they are looking for—and for confidence in the tool to be high, this should also be consistent.

Since CVC4 only returns one result that satisfies all examples, it will successfully synthesize none of the modified benchmarks by construction of the benchmark set. To test RQ1 and RQ2, we implemented a naive best-effort search using CVC4:

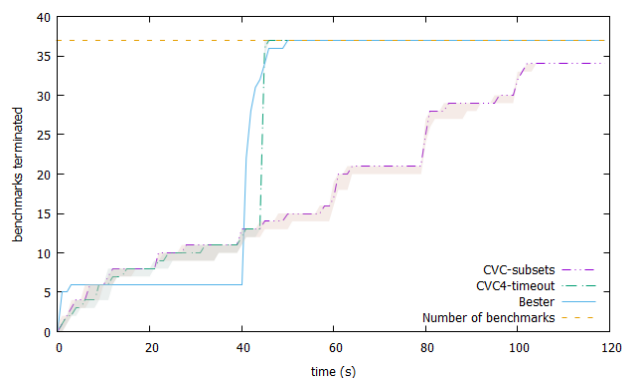
- CVC4-SUBSETS runs on \mathcal{E} , and then on all subsets of size $|\mathcal{E}| - 1$ in a random order. Each such run is done with a 20s timeout (a longer timeout would give BESTER an unfair advantage in the measurements, and as can be seen in Figure 4b, 20s is sufficient for CVC4 for most of the unmodified benchmarks), and results are accumulated in the order that they are discovered and deduplicated in-order.
- CVC4-TIMEOUT runs as CVC4-SUBSETS, but with an additional overall timeout of 45s, in order to be comparable to BESTER.

We ran the 37 modified benchmarks with BESTER, CVC4-SUBSETS, and CVC4-TIMEOUT. Since CVC4-SUBSETS and CVC4-TIMEOUT depend on the random order of the dropped examples, we ran each 5 times and indicate the median and variance. The results are shown in Figure 3.

RQ1: Can BESTER discard contradictions in the example set? Out of 31 benchmarks in the “no-solution” subset of the modified benchmarks, BESTER returned the gold-standard solution first for 26, and the remaining 5 as the second solution. CVC4-SUBSETS returned the gold-standard solution within the top 3 for only 25 of the 31 “no-solution” benchmarks (over 5 runs, min 23, max 28), notably failing completely to synthesize a specification with more than one erroneous example, of which “no-solution” contains two. In addition, it only returned the gold-standard solution first for 17 of the benchmarks (min 16, max 18), with



(a) Number of benchmarks in which the gold-standard solution was returned for a given length of result list. More benchmarks in which a gold-standard solution was found in a shorter list is better. CVC4 runs include a random component, so indicated is the median over 5 runs, with the shaded area indicating the variance.



(b) Number of benchmarks that terminate within a given length of time. This is irrespective of correctness, as the tool must first terminate for its results to be judged by the user. CVC4 runs include a random component, so indicated is the median over 5 runs, with the shaded area indicating the variance. The first plateau for BESTER indicates the “overfitted” benchmark set, where a fully-satisfying but overfitted program is found within the timeout.

■ **Figure 3** Correctness and termination times on benchmarks containing at least one erroneous example.

some gold-standard solutions being as low as fifth. Finally, CVC4-TIMEOUT fails to return a gold-standard solution in the top 5 for 15 of the 31 benchmarks (min 13, max 18), and only returns the gold-standard solution first for 7 of them (min 2, max 9).

We therefore conclude that BESTER is effective at discarding contradictions from the specification and returning a desirable program to the user. Additionally, we conclude that our efficient best-effort implementation is more efficient than a naive approach using a state of the art synthesizer.

RQ2: *Can BESTER return a useful solution despite an overfitted program matching the examples?* Out of the remaining 6 “overfitted” modified benchmarks, BESTER shows 5 in the top three results and 4 in the top 2, exactly the same as CVC4-SUBSETS (min 4, max 6 and min 3, max 4, respectively). CVC4-TIMEOUT had 4 in the top three results (min 3, max 5) and 3 in the top two (min 2, max 4).

We can also see the “overfitted” benchmarks in Figure 3b, as the first plateau between 3 and 40 seconds: overfitted programs are found quickly, and other program options collected along the way are also shown to the user, as opposed to enumerating a benchmark from “no-solution”, which will continue until the timeout.

We conclude that BESTER performs as well as CVC4-SUBSETS and CVC4-TIMEOUT at ranking the gold-standard solution in the top 3 when an overfitted solution exists. This is done more efficiently than a naive solution implemented with CVC4, which still pays the overhead of having to perform multiple runs.

5.2 Partially solving hard benchmarks

In RQ3, we examine the results of BESTER on the “hard” set of benchmarks, which are benchmarks that a simple enumerating OE-reduction synthesizer cannot complete within 40s. BESTER also runs with a timeout of 40s, but returns any best-effort results it finds. None of the results returned will be a gold-standard solution, but they may be part of a path to a solution. Therefore, to answer RQ3, we try to quantify how much of each of the results returned by BESTER can be used to construct a solution.

In order to do that, we must first define the way we measure this similarity.

Tree similarity

In order to judge how much of a result returned by BESTER is relevant to the user, we use a similarity metric between trees on the ASTs of the BESTER result and the gold-standard solution. This metric essentially counts what non-trivial parts of the code can be copied out verbatim.

When computing $s(p_1, p_2)$, we look for maximal sub-expressions (or subtrees) x within p_1 (denotes $x \in p_1$) that are also included in p_2 . For each such x , if $height(x) > 0$ (i.e., x is not a leaf node) we count $terms(x)$. Additionally, we reward the same term for using some identical children even if not *all* children are identical. For example, if there exist two trees, $t(x, y) \in p_1$ and $t(x, z) \in p_2$ (notice that t is the same node type and x is in the same location) we count the root t in addition to $terms(x)$, i.e., add 1 to the accumulated similarity.

Equivalent programs that result in structurally different trees (e.g., `(str.++ "be seeing" (str.++ " " "you"))` vs. `(str.++ (str.++ "be seeing" " ") "you")`) were handled manually by first performing equivalence-preserving tree transformations on the gold standard and then computing the similarity.

Other similarity metrics were originally considered. Program repair projects often employ distance metrics between programs to choose between several possible repairs. Distance metrics for structured objects such as DIFFX [1] for XMLs were applied to ASTs, and application-specific ones were crafted [15, 57]. However, the fragment mapping employed by such distances is more useful for describing insertion and deletion of code (e.g., wrapping a part of the tree in a conditional, removing a statement), whereas we are interested in pieces of code that can be used without modification.

Usable parts of best-effort solutions

We ran BESTER on the 16 benchmarks in the “hard” set. The results are shown in Table 1.

RQ3: *Can BESTER return a useful best-effort solution for tasks that it cannot solve within the timeout?* On average, BESTER results discover over 40% of the gold-standard solution to a task (or the most similar one, if there is more than one), or over 11 terms. When considering

■ **Table 1** Portions of the gold-standard solutions discovered by BESTER for the tasks in the “hard” set. The first set of columns is information on the gold standard solutions available for a task: number and average size. The second set shows the program BESTER ranked first: size, its similarity to the most similar gold-standard solution, and what percentage of the terms in the gold-standard solution is covered ($sim(p,gs)/terms(gs)$). For the closest solution to a gold-standard solution, the rank of the program in BESTER’s list is also indicated. t denotes terms, h denotes height (this is zero-based), sim denotes the similarity to most similar gold-standard solution.

benchmark	gold standard			top BESTER solution				closest solution to GS				
	# GS	avg h	avg t	h	t	sim	% best GS	rank	h	t	sim	% best GS
11604909	3	3.7	15.0	2	12	8	62%	1	2	12	8	62%
30732554	1	3.0	12.0	0	1	0	0%	1	0	1	0	0%
38871714	2	6.0	19.0	2	7	7	37%	1	2	7	7	37%
39060015	2	11.0	72.0	2	6	0	0%	2	2	7	15	45%
41503046	3	8.0	64.7	2	7	11	7%	1	2	7	11	7%
43606446	2	5.5	24.5	3	16	12	38%	1	3	16	12	38%
44789427	3	5.7	40.3	2	7	15	21%	2	2	11	16	73%
bikes	2	4.0	16.5	3	14	13	50%	3	3	14	20	77%
count-total-words	1	5.0	35.0	3	14	22	63%	3	3	20	23	66%
exceljet2	1	7.0	43.0	2	11	14	33%	1	2	11	14	33%
stackoverflow1	1	3.0	16.0	2	9	9	56%	1	2	9	9	56%
stackoverflow2	1	6.0	28.0	3	20	24	86%	1	3	20	24	86%
stackoverflow3	1	4.0	12.0	2	6	0	0%	1	2	6	0	0%
strip-html	1	4.0	15.0	-	-	-	-	-	-	-	-	-
univ_2_short	1	4.0	20.0	2	7	7	35%	1	2	7	7	35%
univ_3_short	1	4.0	14.0	0	1	0	0%	1	0	1	0	0%

only the programs ranked first by BESTER, 32% of the gold-standard solution is discovered with an average of almost 9.5 terms. In 3 of the benchmarks, the entire top-ranking BESTER result was a sub-expression of the solution to the task.

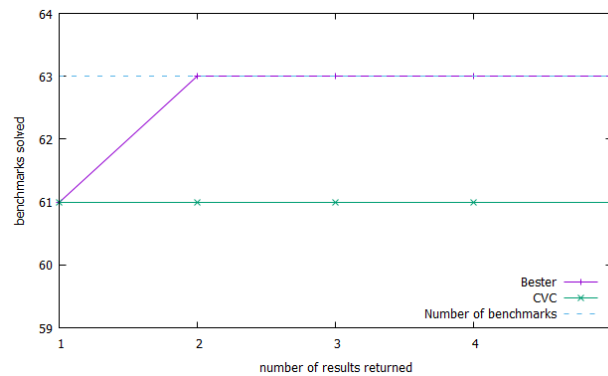
Notice that in some of the tasks (e.g., STACKOVERFLOW2) the similarity between the BESTER result and its nearest gold-standard solution is greater than the number of terms in the BESTER result. This is because an expression in the BESTER result can repeat multiple times in the gold-standard solution.

In one of the 16 benchmarks, BESTER did not find any program that satisfies at least one example, and so returned no programs. In 3 additional benchmarks, none of the programs returned had any non-trivial subtree in common with a gold-standard solution.

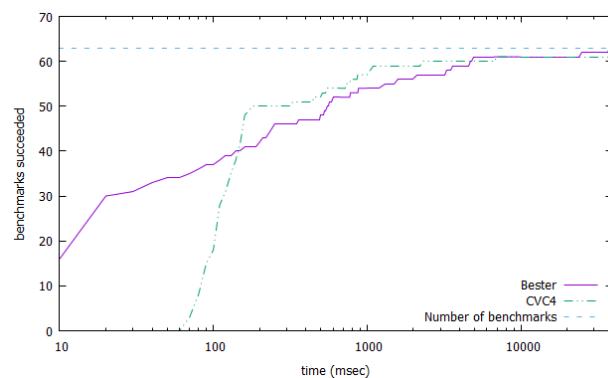
Overall, we conclude that BESTER generates results that can advance the user toward a solution even when they do not fully satisfy the specification. This will be further demonstrated in Section 6. Even though in some of the benchmarks none of the results had any usable components, these are still a minority (overall a quarter of the benchmarks) and the high similarity of those that did succeed indicates the approach can be of great use to a user.

5.3 Solving the original easy benchmarks

Since BESTER ranks its results, taking into account but not relying solely on the number of examples satisfied (see Section 4), we must verify that the solutions to the original, unmodified benchmarks that can be solved by the simple OE synthesizer are still found.



(a) Number of benchmarks in which the gold-standard solution was returned for a given length of result list. CVC4 only returns a single result.



(b) Number of benchmarks that terminate within a given length of time, graph is logscale.

■ **Figure 4** Correctness and time to solution on “easy” benchmarks. CVC4, which was part of the baseline for correct results, is correct every task that terminates within the 40s timeout. CVC4 is faster, but the difference is not extreme.

To test RQ4, we ran BESTER and CVC4 with a 40s timeout on the unmodified “easy” set of benchmarks. The results are in Figure 4.

RQ4: *Can BESTER return the correct result for unmodified “easy” benchmarks?* BESTER succeeds in returning a correct solution that is ranked first for 61 out of 63 of the benchmarks in the “easy” set, on par with the performance of CVC4. (Since CVC4 was used in the creation of the gold standard, it succeeds on every benchmark it terminates on within the 40s timeout.)

In the remaining two benchmarks, the gold standard solution is ranked second. In both of these benchmarks, the desired outputs are a substring of `arg0`, the input variable. Both also contain multiple examples where the input is unchanged. For both of these benchmarks, BESTER ranks the program `arg0` before the target program, since it satisfies some of the examples and is very close to the correct output in the others, uses all the variables, and is very simple. This is rare, and when presented to a user, as in Figure 1a, the program would be accompanied by the number of benchmarks it solves, and we believe it will be easy for users to discard.

Additionally, BESTER is not considerably slower than CVC4 on the benchmarks in “easy”.

We conclude that implementing the best-effort enumeration in BESTER does not harm its correctness on benchmarks that contain no error or contradiction, and that its efficiency in such cases is not much worse than a state of the art synthesizer optimized for competitions.

6 An Exploratory User Study

In this section we detail the results of a small exploratory study in which 8 users were asked to use BESTER to perform two tasks each. Tasks were selected from the benchmark suite presented in Section 5, from the “hard” set of benchmarks, i.e., benchmarks that could not be solved within the timeout by a simple enumerating synthesizer. Notice that these are not modified tasks, i.e., they are identical to their version in the Euphony benchmark set from which they both originated. After completing the tasks, we asked each user to answer a set of questions in a brief interview.

Experiment setup

8 graduate students participated in the study. Users were presented with a brief task description (as it appears in a comment in the benchmark file), the examples in the benchmark, and the grammar at their disposal. As shown in Section 2.1, the semantics of some grammar elements can be misleading, particularly in edge cases.

Participants used a REPL for the target SYGUS language that is initialized with the limited grammar and the example set for the task. For each program entered, the REPL prints the output for every input in the example set. Satisfied examples (matching the example’s expected output) are indicated in green. Screenshots of the REPL are shown in Figure 1. Participants could edit the program on their own or, at any point, call the synthesizer to find a program that would satisfy the examples. The BESTER synthesizer runs either until a timeout of 40s or until interrupted by the user (“press any key” in Figure 1b). While synthesis ran, a number showing the maximum number of examples satisfied was shown and updated when new programs were found. The top 5 programs found by the synthesizer are presented to the user, and can be executed or copied. Participants could call the synthesizer multiple times in the course of one task, as a longer wait could possibly yield more results.

A task was concluded when a participant said they solved the task, when they gave up on the task, or when 20 minutes had elapsed.

Users were told that the tasks are underspecified, and they may resolve any ambiguity as they see fit. Correctness was judged according to semantic equivalence to one of the gold standard solutions for Section 5.

After performing the tasks, users were given a brief structured interview with questions about their use of the synthesizer and the helpfulness of the results. Each participant was paid \$10.

Study tasks

The two tasks given to the participants shared a SYGUS grammar, differing only in the available string literals.

Task 1: “stackoverflow1” in Table 1. Its comment in the benchmark file, provided to participants, was “function to replace substring”.

	arg0	expected result
Examples:	"Trucking Inc."	"Trucking"
	"New Truck Inc"	"New Truck"
	"ABV Trucking Inc, LLC"	"ABV Trucking"

The available string literals were: "", " ", "Inc", ".", ",", and "LLC".

Task 2: “41503046” in Table 1. Its comment in the task file was “find string in substring with lookup”.

	arg0	expected result
Examples:	"Polygonum amphibium"	"Polygonum"
	"Hippuris vulgaris"	"Hippuris"
	"Lysimachia vulgaris"	"Lysimachia"
	"Juncus bulbosus ssp. bulbosus"	"Juncus bulbosus"
	"Lycopus europaeus ssp. europaeus"	"Lycopus europaeus"
	"Nymphaea alba"	"Nymphaea"

The available string literals were: "", " ", and "ssp.".

Research questions

In order to find out whether the *best-effort* paradigm can be useful to programmers, we attempt to answer the following questions:

RQ1: Did users apply any part of the results from BESTER to their solution?

RQ2: Did users find the results from BESTER helpful even though they do not satisfy every example?

6.1 Observed behavior

Participants completed task 1 in an average of 9.56 minutes and task 2 in an average of 11.35 minutes. The fastest solution was programmed in just under 5 minutes.

Of 8 users performing two tasks each, 7 successfully completed both tasks. One user failed to finish the first task within the 20 minute bound and successfully finished the second task. In addition, one user finished the second task with an incorrect result, and, as they were not satisfied with it and had time left, continued to rewrite it until they reached a correct result.

In 15 of the 16 task sessions, the users called the synthesizer at some point during the session. In task 1, 3 of the users ran the synthesizer a second time in the course of the session. In task 2, 2 of the users did so, and one ran the synthesizer a third time. One user performed task 1 without running the synthesizer at all.

Users waited for the synthesizer an average of 17.5s per session while working on task 1 and 27.6s per session while working on task 2, or an average of 14s per individual run of the synthesizer for task 1 and an average of 18.4s for task 2. Only twice did users allow their synthesis request to run until the 40s timeout, both in the course of solving task 2.

7 of the 8 participants executed the top synthesis result once the synthesizer terminated. Only one user executed any result other than the top result – and they executed all results. 6 users later returned to an executed synthesis result using the REPL history and continued to edit it from there.

6 of the users used the mouse to highlight and copy a synthesized expression and paste it into their code. Two users also copied parts of a synthesized expression, but for the most part, the synthesis results that were copied by users were used in their entirety and placed within larger expressions.

Task 1 has two possible modes of solution: one using `str.substr` to slice the string up to the occurrence of "Inc" and using `str.replace` to replace undesirable substrings with "". Four users followed the synthesizer's lead in solving the task with `str.replace`, and another user attempted this and abandoned the direction.

Of the 8 users, 5 ran the synthesizer immediately upon being given task 1 (of the 3 who did not, one did not run the synthesizer at all), and 7 ran it immediately upon being given task 2.

Many of the participants struggled with the behavior of the `str.indexOf` function which returns the index of a substring within a string. Unlike the simplified version included in the grammar in Figure 2a, the function takes an integer parameter which indicates at what index the search for the substring should begin. Many of the users assumed the index parameter to indicate which occurrence of the string should be returned. In the solution of task 2, users spent some time trying to get the second occurrence of " " under this assumption.

6.2 Interviews

In the interview conducted after the tasks were concluded, participants were asked about their decision to call the synthesizer (and to call it again in the course of the session, if they did so), about how they decided how long to wait for the synthesizer, and about the helpfulness of the results.

Calling the synthesizer

Several users explained their call to synthesis as a way to search for a solution they were not seeing, or in hopes it will simply solve the task for them (or, in the case of one user, "just to see what it can do"). Some also recognized, particularly for task 2, that there may be at least a subproblem that can be solved by the synthesizer, providing them with "a start on the solution" or "a piece that can be reused".

However, many of the users explained their call to synthesis as a way to help them understand the problem: either by seeing if there was a generalization of the examples they were not considering, or to get a confirmation of their understanding, "make sure the model in [their] head was correct".

The user who performed task 1 without synthesis said they did not think there exists a simpler way to perform the task than the one they had in mind, so there was no need for synthesis.

Finally, many of the users explained that synthesized code was, to them, a good source of example programs on the inputs. Synthesized code gave them examples of *a*) the language syntax and useful available functions, *b*) the semantics of the functions, and the order of the arguments, *c*) function composition, and how different functions interact, and *d*) help dealing with what one of the users called "an unnatural collection of primitives".

Waiting for the synthesizer

Most users who ran the synthesizer immediately at the start of the task attested that it seemed to them a good use of time to let it run as they were reading the task – it might find something and save them the effort. One user ran the synthesizer again (and to timeout) while they were thinking through a problem they had encountered, just in case.

Users could stop the synthesizer at any time. Three of the users said they used the printout of how many examples were solved by the best discovered program as an indication of when to stop: “[as long as] it made some progress, it was fine”. When the number plateaued, they “figured it solved part of the problem, but the rest isn’t easy.”

Frustration was also a deciding factor in willingness to wait. Users who were not having a hard time with the tasks and simply wanted some reference, terminated the synthesizer very quickly, and they just wanted to see the first results rather than be slowed down by waiting. Users who were more frustrated, especially those who entered task 2 frustrated from task 1, expressed being more willing to wait. The user who failed to finish task 1 and ran the synthesizer to timeout (40s) in task 2 said, “I really struggled, so even if the timeout was 10 minutes, it’s worth it.”

Only two of the 8 users explicitly named impatience as the criterion for deciding how long to wait for the synthesizer.

Half (4) the users re-ran the synthesizer within the course of the same task for one of the two tasks. All said it was in hopes that waiting longer would produce more or better results. One did so because they lost their train of thought and wanted to start over from a synthesized solution in order to remember what they were trying to do, and had forgotten they can call up the solutions from the last run of the synthesizer. This user also stated that, as they were struggling a bit, they were now more willing to wait for a result. Two users stated wanting to utilize time when they had stopped to think about what to do next, in case better solutions would be found. (One user who did not run it a second time said that “in hindsight, letting it run while I was thinking would have been good.”) One user said they were curious as to whether there was a random component that would lead to different results.

Helpfulness of the results

All participants stated the synthesized results were helpful to them in some way.

Getting to a solution: In each of the tasks, the synthesizer returned a different kind of a sub-solution. In task 1, it needed to be wrapped in more function applications to solve more cases, whereas task 2 required a case-split and the synthesizer returned a solution to one of the two cases. Some users viewed one as far more helpful than the other, though which one was not a constant. Some treated the solution to task 1 as “nearly solved the problem”, whereas others saw the solution to task 1 as less helpful but the solution to task 2 as giving them the subprogram that they wanted, where “I could just steal that as a subcomponent”.

Comprehension of the language: Participants who used the synthesizer to understand the language said synthesized results gave them “phrases” for later use and what constants were available; “here is some code, here’s what it does.” (In task 2, when they got used to the language, it was less helpful). Those who did not trust themselves with the language trusted synthesized code.

Comprehension of the task: Users also attested that synthesized results helped them better understand the task itself and in what way the examples generalized. This was particularly true in the second task which contained a case-split. Users said the result of the synthesizer classified the examples for them into the two cases of the split, or as one user said, “once I saw the response from the synthesizer, I knew exactly what the correct answer was.”

6.3 Discussion

We return to our research questions:

RQ1: *Did users use BESTER results in their code?* Participants of our study used both entire BESTER results and subprograms of them in their solution code. In addition, in task 1, several participants let the synthesizer direct the algorithm of their solution. We therefore answer this question in the **affirmative**.

RQ2: *Did users find the results of BESTER helpful?* Participants of our study listed different ways in which the results of BESTER were helpful to them, including (but not limited to) finding code that solves a subproblem. Synthesizer results were also widely used as a comprehension tool by the users. We therefore answer this question in the **affirmative**.

6.4 Threats to validity

Finally, we briefly discuss the threats to the validity of our conclusions from the study.

Number of participants and number of tasks: The study was conducted on 8 participants, performing only two tasks each, which is not enough to make any statistically significant claims. We therefore try to steer away from such conclusions, and instead observe and report usage patterns that occurred throughout user sessions.

Selection of programming language: While using the SYGUS language can be seen as an advantage of the study, mimicking a situation where users are not the most familiar with the language or API they are using, and therefore need the help of a synthesizer, it is also not the easiest programming language to read or write, and includes nontrivial semantics for some of its functions (as demonstrated both in Section 2.1 and in this section). This may lead to different results than a synthesizer for a programming language users are more comfortable reading and writing. All the participants in the study were familiar with the S-expression syntax and had some experience in using it, mitigating some of the comprehension difficulty if not the problematic semantics.

Homogeneous participants: Since students were recruited from a single department in a single institution, there is great similarity in their knowledge and ability. This may have resulted in similar behaviors in the course of the study.

Inability to specify the synthesizer: The BESTER implementation used in the study was not fully-equipped for an iterative and interactive workflow, and users could not control the specifications the synthesizer attempted to solve. This also means users did not spend time on (or have a learning curve in) entering specifications or deciding what they should be. Within such a larger workflow, the observed behaviors may be different. However, we have tried to only draw conclusions about the usefulness of the results of a synthesizer iteration, rather than on the interactive incorporation of synthesis in the development workflow.

7 Related work

Syntax-guided synthesis [3] is the domain of program synthesis where the target program is derived from a set of syntax rules. [30, 55, 18, 56] all fall within this scope. FLASHFILL and FLASHMETA [23, 42] are tools for automating string transformations and data wrangling tasks, whose DSL design centers the delicate balance between an expressive grammar, which

is needed to find a solution, and a tractable enumeration. Padhi et al. [38] raise the issue of the overfitting of an over-expressive grammar, leading to programs such as the one shown in Section 2.1.

The SyGuS competition [5, 4] is held every year and allows solvers and synthesizers to compete for both performance and correctness on a large selection of benchmarks. The competition introduced a PBE track in 2016, and now has two PBE tracks, one for string tasks and one for bit-vector tasks. Both CVC4 [45] and EUSolver [6] have won the competition in the past.

Programming by Example is a popular technique in program synthesis that leverages either user-provided input-outputs [56, 36, 21, 25, 23, 24, 42, 58] or tests [20]. Most PBE techniques target exact specifications and do not handle **noise** in user input. Some notable exceptions are FLASHFILL [23] and RULESYNTH [48], as well as Bayesian and neural program induction techniques [16, 17, 53]. None of these approaches, however, compute results for all subsets of examples, or deal with timeouts.

Ranking and returning multiple results are two common approaches to handling ambiguous specifications in program synthesis; the two often – but not always – go hand-in-hand. The FLASHX tool family [23, 42] uses a ranking function to select a single, most likely program among all the programs that satisfy all user-provided examples. This line of work has explored both hand-crafted [23] and learned [47] ranking functions. Recent work on guiding synthesis using learned probabilistic models [33] can also be seen as applying a learned ranking, but during synthesis rather than at the end. Our ranking function for BESTER is hand-crafted, but is different from existing work in that it incorporates semantic features of programs in addition to syntactic ones, such as the number of examples satisfied, and the distance between the expected and actual outputs. Recent work on synthesizing lenses [35] proposed a novel approach to semantic ranking based on information theory. In the future we would like to explore whether best-effort synthesis can benefit from a more sophisticated ranking function along these lines. Unlike PBE tools, which use ranking to select a single result, code completion tools [28, 43] typically present a ranked list of results to the user, and most commonly rely on learned statistical models and syntactic features.

Observational equivalence Many enumerating synthesizers apply equivalence reductions as a form of pruning the program space [28, 36, 22, 21, 49]. Observational equivalence [2, 55], as a more aggressive and therefore more optimizing form of equivalence, is used in many bottom-up synthesizers [56, 6, 51, 41].

EUSOLVER [6] specializes in solving benchmarks that require case-splitting by performing an OE-reduced enumeration searching for two subprograms that together cover the examples and a condition to decide between them. The enumeration performed by EUSOLVER is similar to that of BESTER in that it is an enumeration over all the examples that also considers subsets of the examples, but only the first program covering a specific subset of the examples is used within the (single) result program, whereas BESTER ranks all such programs and returns the highest ranking ones even if several of them cover the same subset of the examples.

Interaction models for program synthesis are a recent field of research, which has taken two main directions: Modifying specification mechanisms and output formats [40, 13] to make synthesis easier to use and better targeted to specific populations of users. Iterative program synthesis [32, 39, 7] focuses on allowing the user to refine the specification while running the synthesizer after each such refinement, essentially making explicit and improving upon what has been the implicit assumption of all synthesis tools. BESTER is currently situated well within the first direction, but we believe it will also aid the users greatly in an iterative setting.

MaxSAT [10] and **MaxSMT** [9] are the formulation of the satisfiability problem in which certain clauses are marked as *hard constraints* and others as *soft constraints*, and the solver attempts to find an assignment that satisfies all hard constraints while maximizing the number of soft constraints satisfied. Viewing the world through this terminology, we see that previous work has viewed user-provided inputs as hard constraints, and even in work where other soft constraints are available to the user [14], examples are still considered hard constraints. In **BESTER**, all examples are soft constraints and a ranking function is being maximized likening our paradigm to weighted MaxSAT. In Section 4 we suggest a case where there could be additional weights between the specifications.

Opportunistic programming [12, 11] is the programming paradigm in which composite programming tasks are solved by hunting for and joining pieces of existing code from other sources. Projects such as **EXAMPLESTACK** [59] are intended to make the process of importing found code easier. The **BESTER** user study in Section 6 demonstrates synthesis as another method that can provide *pieces* of the solution to the programmer.

8 Conclusion

We proposed a new program synthesis paradigm we call *best-effort program synthesis*, where the synthesizer returns a ranked list of programs that satisfy some part of the specification, rather than just one program that satisfies all of it or no program at all.

This paradigm is implemented in a *best-effort enumeration*, a new synthesis algorithm that extends a bottom-up enumeration with observational equivalence, and is able to accumulate multiple partially-valid results with minimal overhead. We implemented this algorithm in a tool called **BESTER**, and evaluated it on 79 synthesis benchmarks from the **SYGUS** competition and the **EUPHONY** benchmark suite.

Our empirical evaluation showed that best-effort enumeration is more efficient and returns better results than a naive approach to best-effort program synthesis, and that **BESTER** returned useful results even when the specification is flawed or too hard: *i*) for specifications containing an erroneous example, the top three **BESTER** results contained the correct solution, and *ii*) for most hard benchmarks, the top three results contained non-trivial *fragments* of the correct solution. Our user study showed that users apply partially-valid results and *parts* of those results to their code. Additionally, we observed that programmers use the output of the synthesizer for comprehension and not only as a possible part of their solution.

References

- 1 Raihan Al-Ekram, Archana Adma, and Olga Baysal. diffx: an algorithm to detect changes in multi-version xml documents. In *Proceedings of the 2005 conference of the Centre for Advanced Studies on Collaborative research*, pages 1–11. IBM Press, 2005.
- 2 Aws Albarghouthi, Sumit Gulwani, and Zachary Kincaid. Recursive program synthesis. In *International Conference on Computer Aided Verification*, pages 934–950. Springer, 2013.
- 3 Rajeev Alur, Rastislav Bodik, Garvit Juniwal, Milo MK Martin, Mukund Raghothaman, Sanjit A Seshia, Rishabh Singh, Armando Solar-Lezama, Emina Torlak, and Abhishek Udupa. Syntax-guided synthesis. *Dependable Software Systems Engineering*, 40:1–25, 2015.
- 4 Rajeev Alur, Dana Fisman, Rishabh Singh, and Armando Solar-Lezama. Sygus-comp 2017: Results and analysis.
- 5 Rajeev Alur, Dana Fisman, Rishabh Singh, and Armando Solar-Lezama. Sygus-comp 2016: Results and analysis. *arXiv preprint*, 2016. [arXiv:1611.07627](https://arxiv.org/abs/1611.07627).

- 6 Rajeev Alur, Arjun Radhakrishna, and Abhishek Udupa. Scaling enumerative program synthesis via divide and conquer. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 319–336. Springer, 2017.
- 7 Shengwei An, Rishabh Singh, Sasa Misailovic, and Roopsha Samanta. Augmented example-based synthesis using relational perturbation properties. *Proceedings of the ACM on Programming Languages*, 4(POPL):56, 2019.
- 8 Pavol Bielik, Veselin Raychev, and Martin Vechev. Phog: Probabilistic model for code. In Maria Florina Balcan and Kilian Q. Weinberger, editors, *Proceedings of The 33rd International Conference on Machine Learning*, volume 48 of *Proceedings of Machine Learning Research*, pages 2933–2942, New York, New York, USA, 20–22 June 2016. PMLR. URL: <http://proceedings.mlr.press/v48/bielik16.html>.
- 9 Nikolaj Bjørner, Anh-Dung Phan, and Lars Fleckenstein. νz -an optimizing smt solver. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 194–199. Springer, 2015.
- 10 Brian Borchers and Judith Furman. A two-phase exact algorithm for max-sat and weighted max-sat problems. *Journal of Combinatorial Optimization*, 2(4):299–306, 1998.
- 11 Joel Brandt, Philip J Guo, Joel Lewenstein, Mira Dontcheva, and Scott R Klemmer. Two studies of opportunistic programming: interleaving web foraging, learning, and writing code. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 1589–1598. ACM, 2009.
- 12 Joel Brandt, Philip J Guo, Joel Lewenstein, and Scott R Klemmer. Opportunistic programming: How rapid ideation and prototyping occur in practice. In *Proceedings of the 4th international workshop on End-user software engineering*, pages 1–5. ACM, 2008.
- 13 Sarah Chasins. *Democratizing Web Automation: Programming for Social Scientists and Other Domain Experts*. PhD thesis, EECS Department, University of California, Berkeley, October 2019. URL: <http://www2.eecs.berkeley.edu/Pubs/TechRpts/2019/EECS-2019-139.html>.
- 14 Yanju Chen, Ruben Martins, and Yu Feng. Maximal multi-layer specification synthesis. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 602–612, 2019.
- 15 Loris D’Antoni, Roopsha Samanta, and Rishabh Singh. Qlose: Program repair with quantitative objectives. In *Computer Aided Verification - 28th International Conference, CAV 2016, Toronto, ON, Canada, July 17-23, 2016, Proceedings, Part II*, pages 383–401, 2016. doi: 10.1007/978-3-319-41540-6_21.
- 16 Jacob Devlin, Jonathan Uesato, Surya Bhupatiraju, Rishabh Singh, Abdel-rahman Mohamed, and Pushmeet Kohli. Robustfill: Neural program learning under noisy I/O. In *Proceedings of the 34th International Conference on Machine Learning, ICML 2017, Sydney, NSW, Australia, 6-11 August 2017*, pages 990–998, 2017.
- 17 Kevin Ellis, Daniel Ritchie, Armando Solar-Lezama, and Josh Tenenbaum. Learning to infer graphics programs from hand-drawn images. In S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, editors, *Advances in Neural Information Processing Systems 31*, pages 6059–6068. Curran Associates, Inc., 2018. URL: <http://papers.nips.cc/paper/7845-learning-to-infer-graphics-programs-from-hand-drawn-images.pdf>.
- 18 Azadeh Farzan and Victor Nicolet. Modular divide-and-conquer parallelization of nested loops. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2019*, pages 610–624, New York, NY, USA, 2019. ACM. doi:10.1145/3314221.3314612.
- 19 Yu Feng, Ruben Martins, Jacob Van Geffen, Isil Dillig, and Swarat Chaudhuri. Component-based synthesis of table consolidation and transformation tasks from examples. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017, Barcelona, Spain, June 18-23, 2017*, pages 422–436, 2017.
- 20 Yu Feng, Ruben Martins, Yuepeng Wang, Isil Dillig, and Thomas W Reps. Component-based synthesis for complex apis. *ACM SIGPLAN Notices*, 52(1):599–612, 2017.

- 21 John K Feser, Swarat Chaudhuri, and Isil Dillig. Synthesizing data structure transformations from input-output examples. In *ACM SIGPLAN Notices*, volume 50(6), pages 229–239. ACM, 2015.
- 22 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 '16, pages 802–815, New York, NY, USA, 2016. ACM. doi:10.1145/2837614.2837629.
- 23 Sumit Gulwani. Automating string processing in spreadsheets using input-output examples. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '11, pages 317–330, New York, NY, USA, 2011. ACM. doi:10.1145/1926385.1926423.
- 24 Sumit Gulwani. Synthesis from examples: Interaction models and algorithms. In *Symbolic and Numeric Algorithms for Scientific Computing (SYNASC), 2012 14th International Symposium on*, pages 8–14. IEEE, 2012.
- 25 Sumit Gulwani. Programming by examples (and its applications in data wrangling). In Javier Esparza, Orna Grumberg, and Salomon Sickert, editors, *Verification and Synthesis of Correct and Secure Systems*. IOS Press, 2016.
- 26 Sumit Gulwani, Susmit Jha, Ashish Tiwari, and Ramarathnam Venkatesan. Synthesis of loop-free programs. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011, San Jose, CA, USA, June 4-8, 2011*, pages 62–73, 2011. doi:10.1145/1993498.1993506.
- 27 Zheng Guo, Michael James, David Justo, Jiaxiao Zhou, Ziteng Wang, Ranjit Jhala, and Nadia Polikarpova. Program synthesis by type-guided abstraction refinement. In *Principles of programming languages*, page to appear, 2020.
- 28 Tihomir Gvero, Viktor Kuncak, Ivan Kuraj, and Ruzica Piskac. Complete completion using types and weights. In *ACM SIGPLAN Notices*, volume 48(6), pages 27–38. ACM, 2013.
- 29 Jeevana Priya Inala and Rishabh Singh. Webrelate: integrating web data with spreadsheets using examples. *PACMPL*, 2(POPL):2:1–2:28, 2018. doi:10.1145/3158090.
- 30 Shachar Itzhaky, Rohit Singh, Armando Solar-Lezama, Kuat Yessenov, Yongquan Lu, Charles Leiserson, and Rezaul Chowdhury. Deriving divide-and-conquer dynamic programming algorithms using solver-aided transformations. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 145–164. ACM, 2016.
- 31 Vu Le and Sumit Gulwani. FlashExtract: a framework for data extraction by examples. In Michael F. P. O’Boyle and Keshav Pingali, editors, *Proceedings of the 35th Conference on Programming Language Design and Implementation*, page 55. ACM, 2014. doi:10.1145/2594291.2594333.
- 32 Vu Le, Daniel Perelman, Oleksandr Polozov, Mohammad Raza, Abhishek Udupa, and Sumit Gulwani. Interactive program synthesis. *CoRR*, abs/1703.03539, 2017. arXiv:1703.03539.
- 33 Woosuk Lee, Kihong Heo, Rajeev Alur, and Mayur Naik. Accelerating search-based program synthesis using learned probabilistic models. In *ACM SIGPLAN Notices*, volume 53(4), pages 436–449. ACM, 2018.
- 34 Vladimir I Levenshtein. Binary codes capable of correcting deletions, insertions, and reversals. In *Soviet physics doklady*, volume 10(8), pages 707–710, 1966.
- 35 Anders Miltner, Solomon Maina, Kathleen Fisher, Benjamin C. Pierce, David Walker, and Steve Zdancewic. Synthesizing symmetric lenses. *Proc. ACM Program. Lang.*, 3(ICFP), July 2019. doi:10.1145/3341699.
- 36 Peter-Michael Osera and Steve Zdancewic. Type-and-example-directed program synthesis. In *ACM SIGPLAN Notices*, volume 50(6), pages 619–630. ACM, 2015.
- 37 Antti Oulasvirta and Pertti Saariluoma. Surviving task interruptions: Investigating the implications of long-term working memory theory. *International Journal of Human-Computer Studies*, 64(10):941–961, 2006.

- 38 Saswat Padhi, Todd D. Millstein, Aditya V. Nori, and Rahul Sharma. Overfitting in synthesis: Theory and practice. In *Computer Aided Verification - 31st International Conference, CAV 2019, New York City, NY, USA, July 15-18, 2019, Proceedings, Part I*, pages 315–334, 2019. doi:10.1007/978-3-030-25540-4_17.
- 39 Hila Peleg, Shachar Itzhaky, and Sharon Shoham. Abstraction-based interaction model for synthesis. In Isil Dillig and Jens Palsberg, editors, *Verification, Model Checking, and Abstract Interpretation*, pages 382–405, Cham, 2018. Springer International Publishing.
- 40 Hila Peleg, Sharon Shoham, and Eran Yahav. Programming not only by example. In *Proceedings of the 40th International Conference on Software Engineering*, pages 1114–1124. ACM, 2018.
- 41 Phitchaya Mangpo Phothilimthana, Aditya Thakur, Rastislav Bodik, and Dinakar Dhurjati. Scaling up superoptimization. In *ACM SIGARCH Computer Architecture News*, volume 44(2), pages 297–310. ACM, 2016.
- 42 Oleksandr Polozov and Sumit Gulwani. Flashmeta: A framework for inductive program synthesis. *ACM SIGPLAN Notices*, 50(10):107–126, 2015.
- 43 Veselin Raychev, Martin Vechev, and Eran Yahav. Code completion with statistical language models. In *ACM SIGPLAN Notices*, volume 49(6), pages 419–428. ACM, 2014.
- 44 Andrew Reynolds, Haniel Barbosa, Andres Nötzli, Clark W. Barrett, and Cesare Tinelli. cvc4sy: Smart and fast term enumeration for syntax-guided synthesis. In *Computer Aided Verification - 31st International Conference, CAV 2019, New York City, NY, USA, July 15-18, 2019, Proceedings, Part II*, pages 74–83, 2019.
- 45 Andrew Reynolds, Morgan Deters, Viktor Kuncak, Cesare Tinelli, and Clark Barrett. Counterexample-guided quantifier instantiation for synthesis in smt. In *International Conference on Computer Aided Verification*, pages 198–216. Springer, 2015.
- 46 Kensen Shi, Jacob Steinhardt, and Percy Liang. Frangel: Component-based synthesis with control structures. *Proc. ACM Program. Lang.*, 3(POPL), January 2019. doi:10.1145/3290386.
- 47 Rishabh Singh and Sumit Gulwani. Predicting a correct program in programming by example. In *Computer Aided Verification - 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part I*, pages 398–414, 2015. doi:10.1007/978-3-319-21690-4_23.
- 48 Rohit Singh, Venkata Vamsikrishna Meduri, Ahmed K. Elmagarmid, Samuel Madden, Paolo Papotti, Jorge-Arnulfo Quiané-Ruiz, Armando Solar-Lezama, and Nan Tang. Synthesizing entity matching rules by examples. *PVLDB*, 11(2):189–202, 2017. URL: <http://www.vldb.org/pvldb/vol11/p189-singh.pdf>.
- 49 Calvin Smith and Aws Albarghouthi. Program synthesis with equivalence reduction. In *Verification, Model Checking, and Abstract Interpretation - 20th International Conference, VMCAI 2019, Cascais, Portugal, January 13-15, 2019, Proceedings*, pages 24–47, 2019. doi:10.1007/978-3-030-11245-5_2.
- 50 Armando Solar-Lezama. Program sketching. *STTT*, 15(5-6):475–495, 2013. doi:10.1007/s10009-012-0249-7.
- 51 Armando Solar-Lezama, Christopher Grant Jones, and Rastislav Bodik. Sketching concurrent data structures. In *ACM SIGPLAN Notices*, volume 43(6), pages 136–148. ACM, 2008.
- 52 Armando Solar-Lezama, Liviu Tancau, Rastislav Bodik, Sanjit Seshia, and Vijay Saraswat. Combinatorial sketching for finite programs. *ACM SIGOPS Operating Systems Review*, 40(5):404–415, 2006.
- 53 Yonglong Tian, Andrew Luo, Xingyuan Sun, Kevin Ellis, William T. Freeman, Joshua B. Tenenbaum, and Jiajun Wu. Learning to infer and execute 3d shape programs. In *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*, 2019.
- 54 Emina Torlak and Rastislav Bodik. A lightweight symbolic virtual machine for solver-aided host languages. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, Edinburgh, United Kingdom - June 09 - 11, 2014*, page 54, 2014. doi:10.1145/2594291.2594340.

2:30 Perfect Is the Enemy of Good: Best-Effort Program Synthesis

- 55 Abhishek Udupa, Arun Raghavan, Jyotirmoy V Deshmukh, Sela Mador-Haim, Milo MK Martin, and Rajeev Alur. Transit: specifying protocols with concolic snippets. *ACM SIGPLAN Notices*, 48(6):287–296, 2013.
- 56 Chenglong Wang, Alvin Cheung, and Rastislav Bodik. Synthesizing highly expressive sql queries from input-output examples. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 452–466. ACM, 2017.
- 57 Ke Wang, Rishabh Singh, and Zhendong Su. Search, align, and repair: data-driven feedback generation for introductory programming exercises. In *ACM SIGPLAN Notices*, volume 53(4), pages 481–495. ACM, 2018.
- 58 Navid Yaghmazadeh, Xinyu Wang, and Isil Dillig. Automated migration of hierarchical data to relational tables using programming-by-example. *Proc. VLDB Endow.*, 11(5):580–593, January 2018. doi:10.1145/3187009.3177735.
- 59 Tianyi Zhang, Di Yang, Crista Lopes, and Miryung Kirnt. Analyzing and supporting adaptation of online code examples. In *Proceedings of the 41st International Conference on Software Engineering*, pages 316–327. IEEE Press, 2019.

Blame for Null

Abel Nieto 

University of Waterloo, Canada
anietoro@uwaterloo.ca

Marianna Rapoport

University of Waterloo, Canada
mrapoport@uwaterloo.ca

Gregor Richards 

University of Waterloo, Canada
gregor.richards@uwaterloo.ca

Ondřej Lhoták 

University of Waterloo, Canada
olhotak@uwaterloo.ca

Abstract

Multiple modern programming languages, including Kotlin, Scala, Swift, and C#, have type systems where nullability is *explicitly* specified in the types. All of the above also need to interoperate with languages where types remain *implicitly nullable*, like Java. This leads to runtime errors that can manifest in subtle ways. In this paper, we show how to reason about the presence and provenance of such nullability errors using the concept of *blame* from gradual typing. Specifically, we introduce a calculus, λ_{null} , where some terms are typed as *implicitly nullable* and others as *explicitly nullable*. Just like in the original blame calculus of Wadler and Findler, interactions between both kinds of terms are mediated by *casts* with attached *blame labels*, which indicate the origin of errors. On top of λ_{null} , we then create a second calculus, λ_{null}^s , which closely models the interoperability between languages with implicit nullability and languages with explicit nullability, such as Java and Scala. Our main result is a theorem that states that nullability errors in λ_{null}^s can always be blamed on terms with less-precise typing; that is, terms typed as implicitly nullable. By analogy, this would mean that `NullPointerException` in combined Java/Scala programs are always the result of unsoundness in the Java type system. We summarize our result with the slogan *explicitly nullable programs can't be blamed*. All our results are formalized in the Coq proof assistant.

2012 ACM Subject Classification Software and its engineering → General programming languages; Theory of computation → Type theory; Software and its engineering → Interoperability; Theory of computation → Operational semantics

Keywords and phrases nullability, type systems, blame calculus, gradual typing

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2020.3

Supplementary Material ECOOP 2020 Artifact Evaluation approved artifact available at <https://doi.org/10.4230/DARTS.6.2.10>.

Funding This research was supported by the Natural Sciences and Engineering Research Council of Canada and by the Waterloo-Huawei Joint Innovation Lab.

Acknowledgements We would like to thank the anonymous reviewers for their valuable feedback.

1 Introduction

The problem of null pointers has plagued programming languages since 1965 [28]. In languages with null pointers, references may be to valid values, or may be null, which cannot be dereferenced. Attempting to dereference a null reference typically raises a runtime exception in modern, garbage-collected programming languages. This presents a problem for



© Abel Nieto, Marianna Rapoport, Gregor Richards, and Ondřej Lhoták;
licensed under Creative Commons License CC-BY

34th European Conference on Object-Oriented Programming (ECOOP 2020).

Editors: Robert Hirschfeld and Tobias Pape; Article No. 3; pp. 3:1–3:28

Leibniz International Proceedings in Informatics



LIPICs Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany



type soundness and for program maintainability: null is considered a subtype of all reference types, and yet has the interface of none. A number of solutions have been created to address this problem, ranging from type-based solutions [4, 7, 9, 10, 20] to static analyses [24, 30], and from statically sound [10] to heuristic [3].

One type-based solution is to liberate null from its special status as subtype of all reference types. In a language with a null isolated as such, references which are nullable must be explicitly specified as such: the type `T` cannot reference null, but a type such as `T?` (“nullable `T`”, in Kotlin) or `T|Null` (“`T` or `Null`”, in Scala) can. These *explicitly nullable* types must be explicitly verified not to be null before being dereferenced. This adds an extra burden on the programmer to perform such checks, but eliminates *all* null dereference errors if used consistently¹.

Unfortunately, modern programming languages with null often inherit it from connected languages, and this inheritance restricts the scope of nullability. Kotlin, C#, and Swift, for example, all have explicitly nullable types, but due to their interactions with Java, other .NET languages, and Objective-C respectively, may still encounter null dereference errors. For instance, Kotlin [15] has explicitly nullable types, but is designed to be fully compatible with Java. But, Java has *implicitly nullable* types – that is, variables and fields of all reference types may refer to null, unsoundly. As a consequence, even if Kotlin’s own type system perfectly prevents all null dereferences, its interactions with Java will lead to problems.

Luckily, the interaction between languages with differing levels of type soundness has been studied, in the field of gradual typing [22]. In this paper, we apply the principles of gradual typing – and, in particular, the core result that unsoundness can always be correctly blamed on the unsound language – to the problem of interfacing languages with explicit nullability and languages with implicit nullability. We use the context of Scala, which has implemented explicitly nullable types as an optional feature of its in-development next compiler², and Java, which has implicitly nullable reference types.

A sophisticated infrastructure, such as gradual typing’s blame, is needed, because there are several ways that nulls can cause problems. Consider the following snippets of Scala and Java code:

<pre> 1 // Scala 2 class ScalaStringOps { 3 def len(s: String): Int = s.length 4 } 5 6 def main() = { 7 val jso = new JavaStringOps() 8 jso.len(null) 9 jso.nlen() 10 }</pre>	<pre> 1 // Java 2 class JavaStringOps { 3 int len(String s) { 4 return s.length; 5 } 6 7 int nlen() { 8 return new ScalaStringOps().len(null); 9 } 10 }</pre>
---	---

Scala’s line 8 calls the `len` method of Java’s `JavaStringOps`. When importing Java code into Scala, Scala must choose how to represent Java’s implicitly nullable types. Naturally, the Java code might – and in this context, will – fail: Java’s line 4 is unsafe. It’s reasonable to instead try to guarantee that the execution of Scala code will never dereference null. A

¹ Care must be taken to handle the related problem of *uninitialized* or *partially-initialized* objects, which can lead to subtle nullability errors [24, 30].

² <https://dotty.epfl.ch/>

natural assumption is that Scala can assure this by importing all reference types as nullable types. For instance, Java’s `String` is reinterpreted as `String|Null`. This option could be cumbersome for users, but may prevent Scala from raising null errors, as all values from Java must be checked. For practical reasons, most implementations choose instead to unsoundly import `String` as `String`, allowing null dereferences in the “safe” language, but as we will see in the next paragraph, plugging this hole is insufficient to solve the soundness problem anyway. A further problem arises because the interaction between these languages is not one-directional.

Consider Java’s line 8. In this context, Scala’s `ScalaStringOps` is imported into Java, and we have no choice: Its `String` can only reasonably be a `String`, even though Scala `Strings` are not nullable, and Java `Strings` are. With this forced unsound type conversion, Java is free to call `len` with null, causing *Scala* to raise a null dereference error on line 3. But, while the error was raised in Scala code, the *cause* for the problem is Java: Java put a null where it was not suitable. We aim to prove that even when errors occur in Scala code, it is the Java code’s fault.

In gradual typing, “well-typed programs can’t be blamed” [27]. In this work, *explicitly nullable programs can’t be blamed*.

This paper’s contributions are:

- A core calculus, λ_{null} (“lambda null”), that formalizes the essence of type systems with implicit and explicit nullability, like those of Kotlin and Scala. λ_{null} is based on the blame calculus of Wadler and Findler [27].
- A higher-level calculus, λ_{null}^s (“stratified lambda null”), that models the interoperability between languages with implicit nullability and languages with explicit nullability. We can think of λ_{null}^s as a stratified version of λ_{null} , where the implicit and explicit terms are kept separate, but can depend on each other, much like Scala code, which can depend on Java code.
- A metatheory for λ_{null} , consisting of the standard progress and preservation lemmas (Lemmas 5 and 8), as well as blame theorems that characterize how nullability errors can occur in λ_{null} (Theorems 15 and 16).
- A metatheory for λ_{null}^s with two main components. First, a semantics of λ_{null}^s that desugars λ_{null}^s terms as λ_{null} terms. Second, our main result, Theorem 22, which states that nullability errors can always be blamed on terms with less-precise typing; that is, terms typed as implicitly nullable. By analogy, this would mean that `NullPointerException`s in combined Java/Scala programs are always the result of unsoundness in the Java type system, which treats reference types as implicitly nullable. In the style of Wadler and Findler [27], we summarize our result with the slogan *explicitly nullable programs can’t be blamed*.
- A Coq mechanization of all our results.

2 Blame Calculus

The blame calculus of Wadler and Findler [27] models the interactions between less-precisely and more-precisely typed code. For example, the less-precisely typed code could come from a dynamically-typed language, and the more-precisely typed code could come from a statically-typed language like Scala. The goal of the calculus is twofold:

- To characterize situations where errors can or cannot occur as a result of the interaction between both languages: e.g. “there will not be runtime errors, unless the typed code calls the untyped code”.
- If runtime errors do occur, to assign *blame* (responsibility) for the error to some term present in the evaluation.

3:4 Blame for Null

To do the above, the blame calculus extends the simply-typed lambda calculus with *casts* that contain *blame labels*³. The notation⁴ for casting a term s from a type S to another type T with blame label p is $s : S \Longrightarrow^p T$.

During evaluation, a cast might succeed, fail, or be *broken up into further casts*. For example, suppose that we cast the value 4 from an integer into a natural number. Such a cast would naturally succeed, and one step of evaluation then makes the cast disappear: $4 : \text{Int} \Longrightarrow^p \text{Nat} \mapsto 4$. A cast can also fail. This is when we use the blame label. For example, if we try to turn an integer into a string using a cast with blame label p , then we fail and blame p : $4 : \text{Int} \Longrightarrow^p \text{String} \mapsto \uparrow p$.

If the cast is *higher-order*, however, things get tricky. How are we to determine whether a function of type $\text{Int} \rightarrow \text{Int}$ also has type $\text{Nat} \rightarrow \text{Nat}$?

$$(\lambda(x : \text{Int}).x - 2) : \text{Int} \rightarrow \text{Int} \Longrightarrow^p \text{Nat} \rightarrow \text{Nat}$$

Informally, the cast above is saying: “if you provide as input a Nat that is *also* an Int , the function will return an Int that is *also* a Nat ”. Intuitively, the cast is incorrect, because the function can return negative numbers. In general, however, we cannot hope to statically ascertain the validity of a higher-order cast. The insight about what to do here comes from work on higher-order contracts [11]. The key idea is to *delay* the evaluation of the cast *until the function is applied*. That is, we consider the entire term above, the lambda plus its cast, a *value*. Then, if we need to apply the lambda wrapped in a cast, we use the following rule:

$$((v : (A \rightarrow B) \Longrightarrow^p (A' \rightarrow B')) w) \mapsto (v (w : A' \Longrightarrow^{\bar{p}} A)) : B' \Longrightarrow^p B$$

Notice how the original cast was decomposed into two separate casts on subterms. This rule says that applying a lambda wrapped in a cast involves three steps:

- First, we cast the argument w , which is expected to have type A' , to type A .
- Then we apply the function v to its argument, as usual.
- Finally, we cast the result of the application from B' back to the expected type B .

Also notice how the blame label in the cast $w : A' \Longrightarrow^{\bar{p}} A$ changed from p to its *complement* \bar{p} . We can think of blame labels as opaque identifiers. We assume the existence of a *complement* function on blame labels, and write \bar{p} for the label that is the complement of blame label p . The complement operation is *involutionary*, meaning that it is its own inverse: $\bar{\bar{p}} = p$.

When a runtime error happens, complementing blame labels leads to *two* kinds of blame: *positive* and *negative*:

Positive blame. Given a cast with blame label p , positive blame happens when the term *inside* the cast is responsible for the failure. In this case, the (failed) term will evaluate to $\uparrow p$. For example, recall our example with the faulty function that subtracts two from its argument:

$$\begin{aligned} & ((\lambda(x : \text{Int}).x - 2) : \text{Int} \rightarrow \text{Int} \Longrightarrow^p \text{Nat} \rightarrow \text{Nat}) 1 \\ & \mapsto ((\lambda(x : \text{Int}).x - 2) (1 : \text{Nat} \Longrightarrow^{\bar{p}} \text{Int})) : \text{Int} \Longrightarrow^p \text{Nat}) \\ & \mapsto ((\lambda(x : \text{Int}).x - 2) 1) : \text{Int} \Longrightarrow^p \text{Nat} \\ & \mapsto (1 - 2) : \text{Int} \Longrightarrow^p \text{Nat} \\ & \mapsto -1 : \text{Int} \Longrightarrow^p \text{Nat} \\ & \mapsto \uparrow p \end{aligned}$$

³ The original presentation in Wadler and Findler [27] also adds *refinement types*, but we will not need them here.

⁴ The notation for casts we use comes from Ahmed et al. [1].

The term being cast (the lambda) is responsible for the failure, because it promised to return a `Nat`, which `-1` is not.

Negative blame. If the cast fails because it is provided an argument of an incorrect type by its *context* (surrounding code), then we will say the failure has negative blame. In this case, the term will evaluate to $\uparrow \bar{p}$. For example, suppose our example function is used in an untyped context, where the only type is `*`. Without help from its type system, the context might try to pass in a `String` as argument:

$$\begin{aligned} & ((\lambda(x: \text{Int}).x - 2) : \text{Int} \rightarrow \text{Int} \Longrightarrow^P * \rightarrow *) \text{"one"} \\ \mapsto & ((\lambda(x: \text{Int}).x - 2) (\text{"one"} : * \Longrightarrow^{\bar{p}} \text{Int})) : \text{Int} \Longrightarrow^P * \\ \mapsto & \uparrow \bar{p} \end{aligned}$$

Because the context tried to pass an argument that is not an `Int`, we blame the failure on the context.

2.1 Well-typed Programs Can't Be Blamed

The central result in Wadler and Findler [27] is a *blame theorem* that provides two guarantees:

- Casts from less-precise⁵ to more-precise types, like $v : \text{Int} \rightarrow \text{Int} \Longrightarrow^P \text{Nat} \rightarrow \text{Nat}$, only fail with *positive* blame.
- Casts from more-precise to less-precise types, like $v : \text{Int} \rightarrow \text{Int} \Longrightarrow^P * \rightarrow *$, only fail with *negative* blame.

In both cases, the less precisely typed code is assigned responsibility for the failure. The authors summarize this result with the slogan “well-typed programs can't be blamed”, itself a riff on an earlier catchphrase, “well-typed programs cannot go wrong”, by Milner [18]. In the next section, we will show how we can adapt ideas from the blame calculus to reason about nullability errors.

3 Main Ideas

This section offers a bird's-eye view of the rest of the paper. The main idea is to cast (no pun intended) the null interoperability problem as a gradual typing problem. Then, using casts with blame, we show that the implicit language can always be blamed for interoperability errors. That is, *explicitly nullable programs can't be blamed*.

3.1 λ_{null}

The first step is to formalize null pointer exceptions. We start with a calculus λ_{null} (“lambda null”), based on the blame calculus of Wadler and Findler [27], to which we add a `null` literal with type `Null`. We keep the casts with blame: $s : S \Longrightarrow^P T$. Additionally, we distinguish between three kinds of function types:

- $\#(S \rightarrow T)$ is a *presumed non-nullable* function, meaning that values of this type are expected to be *non-null*, but *could* be `null` if a downcast was involved (see Section 4). That these functions should be non-null is relevant to how we assign blame.
- $?(S \rightarrow T)$ is a *safe nullable* function, meaning that values of this type *can* be `null`, but the type system makes sure that they are safely used.
- $!(S \rightarrow T)$ is an *unsafe nullable* function, meaning that values of this type *can* be `null`, but the type system does not protect against unsafe uses of them.

⁵ The formal definition of “less-precise” is given by a *naive subtyping* relation in Wadler and Findler [27].

3:6 Blame for Null

The table below shows the three function types in λ_{null} and the kinds of Java and Scala types they model⁶:

λ_{null}^s	Scala	Java
$\#(S \rightarrow T)$	<code>String_{Scala}</code>	
$?(S \rightarrow T)$	<code>String!Null</code>	
$!(S \rightarrow T)$		<code>String_{Java}</code>

Nullability errors happen when we have a function application $u v$, but the value u in the function position is in fact `null`. This corresponds closely to what happens in real languages, where null pointer exceptions occur when we select a field or method on a `null` receiver: e.g. we evaluate `s.length()` and `s` is `null`. In fact, u will be “disguised” inside one or more casts, so the type system is fooled into thinking u is a function. For example, taking one step of evaluation on the following term leads to an error $\uparrow \bar{p}$, where the label in the error comes from the cast: $(\text{null} : \text{Null} \Rightarrow^{p?}(\text{Null} \rightarrow \text{Null})) \text{null} \mapsto \uparrow \bar{p}$.

If one wants to be safe from nullability errors, then instead of a regular application $s t$, we can use a *safe application* `app(s, t, r)`, which conceptually desugars into `if (s != null) then (s t) else r`.

3.2 Blame Assignment

In the example above, $(\text{null} : \text{Null} \Rightarrow^{p?}(\text{Null} \rightarrow \text{Null})) \text{null} \mapsto \uparrow \bar{p}$, how did we decide to blame \bar{p} ? The basic rules for assigning blame are as follows:

- If the cast that causes the failure casts to a *presumed non-nullable* function, e.g. $v : ?(S \rightarrow T) \Rightarrow^p \#(S \rightarrow T)$, then we blame the *cast*: i.e. $\uparrow p$. This is because the context (the surrounding code) was promised a value that should *not* be `null`, yet the cast delivered `null`.
- On the other hand, if the cast is to an *unsafe nullable function*, e.g. $v : \#(S \rightarrow T) \Rightarrow^{p!}(S \rightarrow T)$, then we blame the context, because the context should know that the presumptive function value could in fact be `null`, but nevertheless chose to use a regular application, instead of a safe application.
- Casts to a *safe nullable function*, e.g. $v : \#(S \rightarrow T) \Rightarrow^{p?}(S \rightarrow T)$, will never fail, because the type system ensures that such functions are *always* applied through safe applications.

In addition to the rules above, our blame assignment needs to support *nested casts*. For example, suppose we have a `null` value that passes through the following casts, $\text{Null} \Rightarrow^p ? \Rightarrow^q \# \Rightarrow^r !$ ⁷. If the resulting cast is used in the function position of an application, it will lead to a failure, but which cast should we blame? We could blame \bar{r} , as per the second blame assignment rule above. However, something feels off, because intuitively a cast $\# \Rightarrow^r !$ should never be blamed for a failure. Indeed, the cast was promised a non-null value, which it should be safe to consider as a `!`. Instead, we identify $? \Rightarrow^q \#$ as the problem, and blame q , as per the first rule above.

To summarize, blame assignment is a two-part process: we first identify the cast responsible for the error using a *blame assignment* relation \uparrow (this might involve skipping over one or more nested casts), and then we blame the relevant label, or its complement, depending on whether the destination type is `#` or `!`.

⁶ Since λ_{null} is a core calculus, it does not have objects or classes, but only functions. In λ_{null} it is function types that are nullable or non-nullable.

⁷ Here we are using a shorthand syntax for casts, where we only show the top-level function type. For example, we abbreviate a cast $s : \#(S \rightarrow T) \Rightarrow^{p!}(S \rightarrow T)$ as $\# \Rightarrow^p !$.

3.3 λ_{null}^s

With λ_{null} sketched, we then define a *second, higher-level* calculus λ_{null}^s (“stratified lambda null”). Whereas in λ_{null} the three function types can be mixed freely, λ_{null}^s stratifies terms into *implicit* and *explicit* sublanguages. Within the implicit sublanguage, we can only use unsafe nullable functions (e.g. $!(S \rightarrow T)$), while in the explicit sublanguage we can use both non-nullable ($\#(S \rightarrow T)$) and safe nullable functions ($?(S \rightarrow T)$). The implicit sublanguage models languages where `null` is a subtype of any other (reference) type, like Java. The explicit sublanguage models languages where the user can choose whether a type is nullable or not, like Kotlin and Scala.

The last step is to model the interoperability between the implicit and explicit worlds. To do that, we add to λ_{null}^s an `import` term that makes an implicit term available to the explicit world and vice versa. Imports look very similar to let-bindings: `importe x : Te = (ti : Ti) in te`. This says that we evaluate the implicit term t_i and assign it to x , which is then available in the body t_e (implicit and explicit terms and types are written in red and blue, respectively). Additionally, the implicit type of t_i is T_i , but to the explicit world the type is translated as T_e . This kind of view shift in the type closely models what happens in real-world languages that support explicit nulls, but need to operate with another language where `null` is implicit. For example, the Java type `String` is translated as `String!Null` in Scala.

3.3.1 Semantics

We give type systems for λ_{null} and λ_{null}^s , and an operational semantics for λ_{null} . The semantics of λ_{null}^s are given via a desugaring to λ_{null} . The desugaring is straightforward, but it allows us to identify the three kinds of casts that can make a program fail:

- *Internal* casts within the *implicit* world.
- *Internal* casts within the *explicit* world.
- *Interoperability* casts that result from desugaring `imports`. For example, the `import` term above generates the cast $t_i : T_i \Longrightarrow^{\mathcal{I}} T_e$. Similarly, an import of an explicit term into the implicit world would generate a cast $t_e : T_e \Longrightarrow^{\mathcal{E}} T_i$. Here, \mathcal{I} and \mathcal{E} are labels that interoperability casts based on the cast’s “direction”.

3.3.2 Metatheory

We show that if we start with a well-typed term from λ_{null}^s , desugar it, and evaluate it using the λ_{null} operational semantics, then the term’s normal form (if it exists) is either a value, or an error with blame. In fact, we are able to characterize this behaviour more precisely. By reasoning about which casts are safe using *positive* and *negative* subtyping, which are standard tools from gradual typing, we are able to show our main result:

- Internal casts within the explicit world can *never* be blamed for failures.
- Interoperability casts *can* be blamed, but we always blame the implicit world in such cases. That is, the blame always goes to \mathcal{I} or $\bar{\mathcal{E}}$.

This main result formalizes our intuition that *explicitly nullable programs can’t be blamed*. It is also evidence that gradual typing can accurately model the null interoperability problem. All our results have been verified in Coq.

3:8 Blame for Null

x, y, z	Variables	$r ::=$	Results
		t	term
p, q	Blame labels	$\uparrow p$	blame
\bar{p}	Label complement		
		$S, T, U ::=$	Types
$f, s, t ::=$	Terms	Null	null type
x	variable	$\alpha (S \rightarrow T)$	function type with modality
null	null literal		
$\lambda(x: T).s$	abstraction	$\alpha, \beta ::=$	Function Type Modality
$s t$	application	$\#$	presumed non-nullable
$\text{app}(f, s, t)$	safe application	$?$	safe nullable
$s : S \Longrightarrow^p T$	cast	$!$	unsafe nullable
$u, v ::=$	Values		
$\lambda(x: T).s$	abstraction		
null	null literal		
$v : S \Longrightarrow^p T$	cast		

■ **Figure 1** Terms and types of λ_{null} .

4 A Calculus with Implicit and Explicit Nulls

In this section, we describe the λ_{null} calculus in full. λ_{null} is based on the blame calculus of Wadler et al. [27, 26]. λ_{null} contains the two key ingredients we need to model language interoperability with respect to **null**:

- Types that are *implicitly* nullable and types that are *explicitly* nullable.
- *Casts* that mediate the interaction between the types above, along with *blame labels* to track responsibility for failures, should they occur.

The terms and types of λ_{null} are shown in Figure 1, and are explained below. Section 5 shows how to use λ_{null} to model the interaction between two languages, each treating nullability differently (like Java and Scala). This section focuses on λ_{null} and its metatheory.

4.1 Values of λ_{null}

A value in λ_{null} can be any of the following: an abstraction $\lambda(x: T).s$, the **null** literal, or another value v wrapped in a cast, $v : S \Longrightarrow^p T$.

The motivation for classifying certain casts as values is as follows. Consider the cast $\text{null} : \text{Null} \Longrightarrow^p!(S \rightarrow T)$. As we will see later, $!(S \rightarrow T)$ is an *unsafe nullable* function type, so the cast can fail. However, the cast does not fail immediately; instead, the cast only fails if *we try to apply* the (**null**) function to an argument, like so $(\text{null} : \text{Null} \Longrightarrow^p!(S \rightarrow T)) w$. This matches e.g. Java’s behaviour, where passing a **null** when an object is expected only triggers an exception if we try to select a field or method from the **null** object:

```
String s = null; // no exception is raised here
s.length()      // an exception is raised only when we try to select a method or field
```

4.2 Terms of λ_{null}

A term of λ_{null} is either a variable x , the literal `null`, an abstraction $\lambda(x : T).s$, an application $s t$, a *safe application* $\text{app}(s, t, u)$, or a cast $s : S \Longrightarrow^p T$. The meaning of most terms is standard; the interesting ones are explained below:

- The `null` literal is useful for modelling null pointer exceptions. Specifically, an application $s t$, where s reduces to `null`, results in a failure.
- A safe application $\text{app}(s, t, u)$ is a regular application that can also handle the case where s is `null`. If s is non-null, then the safe application behaves like the regular application $s t$. However, if s is `null` then the entire safe application reduces to u . Safe applications could be desugared into a combination of if-expressions and flow typing [12]:

$$\text{app}(s, t, u) \equiv \text{if } (s \neq \text{null}) \text{ then } s t \text{ else } u$$

In particular, this means safe applications are “lazy”: they do not initially evaluate either the argument t or sentinel value u . Instead, we only evaluate the expression s in function position, and then proceed depending on whether s is `null` or not.

For the desugaring above to work we would need flow typing, because within the `then` branch we need to be able to assume that s is non-null. Safe applications allow us to work with nullable values without introducing flow typing.

Safe applications closely model Kotlin’s “Elvis” operator [16], written `?:`. In Kotlin, the expression `a ?: b` evaluates to `a`, unless the left-hand side is `null`, in which case the entire expression evaluates to `b`.

- The cast $s : S \Longrightarrow^p T$ is used to change the type of s from S to T . The blame label p will be used to assign blame should the cast cause a failure.

Finally, the *result* of evaluating a λ_{null} term is either a value v or an error with blame p , denoted by $\uparrow p$.

4.3 Types of λ_{null}

The types of λ_{null} are also shown in Figure 1. There are four kinds of types:

- The `Null` type contains a single element: `null`.
- The *presumed non-nullable* function type $\#(S \rightarrow T)$, as the name indicates, contains values that *should not* be `null`. However, the value might *still* end up being `null`, through casts. This corresponds to non-nullable types like `StringScala`. For conciseness, we will refer to these types simply as *non-nullable* function types.
- A value with *safe nullable* function type $?(S \rightarrow T)$ is allowed to be `null`. The type system will ensure that any such functions are applied using safe applications. This corresponds to nullable union types like `StringScala | Null`.
- By contrast, a value with *unsafe nullable* function type $!(S \rightarrow T)$ is also allowed to be `null`, but the type system does not enforce a null check before an application. That is, if s has type $!(S \rightarrow T)$, the type system will allow both $s t$ and $\text{app}(s, t, u)$, even though the former might fail. This corresponds to types in Java, which are implicitly nullable.

As we will see below, some typing rules apply to more than one function type. For example, when typing an application $s t$, we will require that s have a type of the form $\#(S \rightarrow T)$ or $!(S \rightarrow T)$. Instead of duplicating the relevant inference rule, the syntax for function types $\alpha (S \rightarrow T)$ includes a *modality* α . In the application case, we can then say that s must have type $\alpha (S \rightarrow T)$ with $\alpha \in \{\#, !\}$.

$\Gamma \vdash t : T$	$S \rightsquigarrow T$
$\frac{\Gamma(x) = T}{\Gamma \vdash x : T} \quad (\text{T-VAR})$	$\text{Null} \rightsquigarrow \text{Null} \quad (\text{C-NULREFL})$
$\Gamma \vdash \text{null} : \text{Null} \quad (\text{T-NULL})$	$\frac{\alpha \in \{?, !\}}{\text{Null} \rightsquigarrow \alpha(S \rightarrow T)} \quad (\text{C-NULL})$
$\frac{\Gamma, x : S \vdash s : T}{\Gamma \vdash \lambda(x : S).s : \#(S \rightarrow T)} \quad (\text{T-ABS})$	$\frac{S' \rightsquigarrow S \quad T \rightsquigarrow T' \quad \alpha, \beta \in \{\#, ?, !\}}{\alpha(S \rightarrow T) \rightsquigarrow \beta(S' \rightarrow T')} \quad (\text{C-ARROW})$
$\frac{\Gamma \vdash s : \alpha(S \rightarrow T) \quad \alpha \in \{\#, !\} \quad \Gamma \vdash t : S}{\Gamma \vdash s t : T} \quad (\text{T-APP})$	
$\frac{\Gamma \vdash f : \alpha(S \rightarrow T) \quad \alpha \in \{?, !\} \quad \Gamma \vdash s : S \quad \Gamma \vdash t : T}{\Gamma \vdash \text{app}(f, s, t) : T} \quad (\text{T-SAFEAPP})$	
$\frac{\Gamma \vdash s : S \quad S \rightsquigarrow T}{\Gamma \vdash (s : S \Longrightarrow^p T) : T} \quad (\text{T-CAST})$	

■ **Figure 2** Typing and compatibility rules of λ_{null} .

Keeping λ_{null} simple. We could reduce the number of function types and avoid the need for safe applications through a combination of sum types and case analysis. For example, in Scala nullable values are represented with sum types (e.g. a nullable string has type `String | Null`). The case analysis in turn requires support for flow-typing:

```
val s : String | Null = ...
// s inferred to have type String in the 'then' branch, so s.length is type-correct
val len : Int = if (s != null) s.length else 0
```

Since λ_{null} is a core calculus, we focus on modelling the assignment of blame for nullability errors, which revolves around blaming casts or their client code, at function application time. This is why λ_{null} eschews sum types and flow typing in favour of primitives for nullable function types and safe applications. Additionally, both of these primitives appear in modern programming languages (e.g. in Kotlin).

4.4 Typing λ_{null}

The typing rules for λ_{null} are shown in Figure 2. The three interesting rules are T-App, T-SafeApp, and T-Cast:

- **(T-App)** The rule for a type application $s t$ is *almost* standard, except that s can not only have type $\#(S \rightarrow T)$, but *also* the *unsafe nullable* function type $!(S \rightarrow T)$. This models languages with implicit nullability (like Java), where the type system allows operations that can lead to null-related errors.
- **(T-SafeApp)** To type a safe application $\text{app}(f, s, t)$, we check that f is a nullable function type; that is, it must have type $?(S \rightarrow T)$ *or* $!(S \rightarrow T)$ (if f had type $\#(S \rightarrow T)$ we would use T-App). Notice that the type of s must be S (the argument type), but t must have type T (the return type). This is because t is the “default” value that we return if f is `null`.

$$\begin{array}{ccc}
\boxed{\mathbf{null}(v)} & & \boxed{\mathbf{abs}(v)} \\
\mathbf{null}(\mathbf{null}) & \text{(N-NULL)} & \mathbf{abs}(\lambda(x:T).s) & \text{(A-ABS)} \\
\frac{\mathbf{null}(v)}{\mathbf{null}(v : S \Longrightarrow^p T)} & \text{(N-CAST)} & \frac{\mathbf{abs}(v)}{\mathbf{abs}(v : S \Longrightarrow^p T)} & \text{(A-CAST)}
\end{array}$$

■ **Figure 3** \mathbf{abs} and \mathbf{null} predicates.

- **(T-Cast)** To type a cast $s : S \Longrightarrow^p T$ we check that s indeed has the source type S . The entire cast then has type T . Additionally, we make sure that S and T are *compatible*, written $S \rightsquigarrow T$. Type compatibility is described below.

Notice that the type of \mathbf{null} is always \mathbf{Null} , so in order to get a nullable function we need to use casts. For instance,

$$\frac{\text{T-NULL} \quad \frac{}{\vdash \mathbf{null} : \mathbf{Null}} \quad \frac{}{\mathbf{Null} \rightsquigarrow ?(\mathbf{Null} \rightarrow \mathbf{Null})} \quad \text{C-NULL}}{\vdash \mathbf{null} : \mathbf{Null} \Longrightarrow^p ?(\mathbf{Null} \rightarrow \mathbf{Null}) : ?(\mathbf{Null} \rightarrow \mathbf{Null})} \text{T-CAST}$$

4.4.1 Compatibility

Compatibility is a binary relation on types that is used to limit (albeit only slightly) which casts are valid. Given types S and T , we can cast S to T only if $S \rightsquigarrow T$. The compatibility rules are shown in Figure 2.

► **Lemma 1.** *Compatibility is reflexive, but is neither symmetric nor transitive.*

A counter-example to symmetry is that $\mathbf{Null} \rightsquigarrow ?(\mathbf{Null} \rightarrow \mathbf{Null})$, but the latter is not compatible with the former. A counter-example to transitivity is that $\mathbf{Null} \rightsquigarrow ?(\mathbf{Null} \rightarrow \mathbf{Null})$ and $?(\mathbf{Null} \rightarrow \mathbf{Null}) \rightsquigarrow \#(\mathbf{Null} \rightarrow \mathbf{Null})$, but \mathbf{Null} is not compatible with $\#(\mathbf{Null} \rightarrow \mathbf{Null})$.

4.5 Semantics of $\lambda_{\mathbf{null}}$

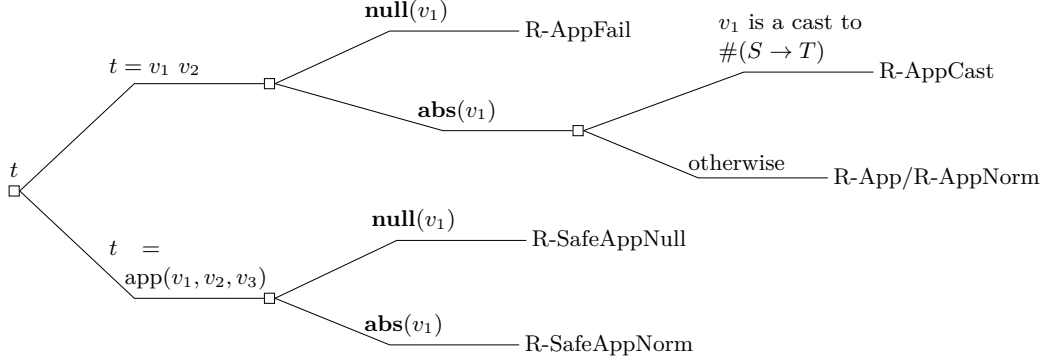
We give a small-step operational semantics for $\lambda_{\mathbf{null}}$, using evaluation contexts. The rules are shown in Figure 5. Notice that the result r of an evaluation step can be a term *or* an error, denoted by $\uparrow p$.

4.5.1 Auxiliary Predicates

The unary predicates on types \mathbf{null} and \mathbf{abs} , shown in Figure 3, test whether a value v is equal to \mathbf{null} or to a lambda abstraction, respectively. These predicates are able to “see through” casts.

► **Example 2.** The following hold:

- $\mathbf{null}(\mathbf{null})$, $\mathbf{null}(\mathbf{null} : \mathbf{Null} \Longrightarrow^p \#(\mathbf{Null} \rightarrow \mathbf{Null}))$
- $\mathbf{abs}(\lambda(x:\mathbf{Null}).x)$, $\mathbf{abs}(\lambda(x:\mathbf{Null}).x : \#(\mathbf{Null} \rightarrow \mathbf{Null}) \Longrightarrow^p ?(\mathbf{Null} \rightarrow \mathbf{Null}))$



■ **Figure 4** Simplified decision tree for λ_{null} reduction rules.

4.5.2 Reduction Relation

The decision tree in Figure 4 shows a simplified view of the reduction rules. The rules are described in detail below.

- **R-App** is standard beta reduction.
- **R-AppFail** handles the case where we have a function application and the value in the function position is in fact `null`. This last fact is checked via the auxiliary predicate `null(v)`. In this case, the entire term (and not just the subterm within the evaluation context) evaluates to an error. What remains is to determine the blame label that we will use. This we do using the *blame assignment* relation (also shown in Figure 5): we write $v \uparrow p$ to indicate that the blame should go to a label p . As we will see in Section 4.5.3, v will contain one or more casts, and the label p is obtained from one of the casts. Here is a sample application of **R-AppFail**, where $v = \text{null} : \text{Null} \Longrightarrow^p!(\text{Null} \rightarrow \text{Null})$:

$$(\text{null} : \text{Null} \Longrightarrow^p!(\text{Null} \rightarrow \text{Null})) \text{ null} \mapsto \uparrow \bar{p}$$

In this case, the only cast in v is selected as the source of the blame (in general, v could contain multiple casts). We blame \bar{p} because the surrounding code (the code doing the application $v \text{ null}$), should have used a safe application, based on v 's nullable type.

- **R-AppCast** handles the case where the value v' in the function position is a cast involving only non-nullable function types; i.e. $v' = v : \#(S_1 \rightarrow S_2) \Longrightarrow^p \#(T_1 \rightarrow T_2)$. In this case, the application $v' u$ reduces to

$$(v (u : T_1 \Longrightarrow^{\bar{p}} S_1)) : S_2 \Longrightarrow^p T_2$$

This is the classic behaviour of blame in a function application, and comes from [11]. The type system guarantees that the argument u is typed as a T_1 , but the function v expects it to have type S_1 . We then need the cast $u : T_1 \Longrightarrow^{\bar{p}} S_1$ before passing the argument to function. Notice that the blame label has been complemented (\bar{p}), because it is the *context* (the code calling the function v) who is responsible for passing an argument of the right type. Conversely, when the function v returns, its return value will have type S_2 , but the surrounding code is expecting a value of type T_2 . We then need to cast the entire application from S_2 to T_2 ; this time, the blame label is p . As Findler and Felleisen [11] remark, the handling of the blame label matches the rule for function subtyping present in other system, where the argument and return type must be contra- and covariant, respectively.

- **R-AppNorm** handles the case where we have an application $v u$, and v is a cast to a nullable function type (either a $?$ function or a $!$ function). Additionally, we know that $\mathbf{abs}(v)$ holds. In this case, what we would want to do is “translate” the nullable function type into a *non-nullable* function type. This is fine because $\mathbf{abs}(v)$ implies that the underlying function is non-null. The *normalization* relation $v \gg v'$ (also shown in Figure 5) achieves this translation of casts.

► **Example 3.** Let $t = \lambda(x: \text{Null}).x$. Suppose we are evaluating the application

$$(t : \#(\text{Null} \rightarrow \text{Null}) \Longrightarrow^{p?}(\text{Null} \rightarrow \text{Null})) \text{ null}$$

We proceed by first noticing that $\mathbf{abs}(t : \#(\text{Null} \rightarrow \text{Null}) \Longrightarrow^{p?}(\text{Null} \rightarrow \text{Null}))$. Then we normalize the value in the function position

$$\frac{\frac{}{t \gg t} \text{ NORM-ABS}}{t : \#(\text{Null} \rightarrow \text{Null}) \Longrightarrow^{p?}(\text{Null} \rightarrow \text{Null}) \gg t : \#(\text{Null} \rightarrow \text{Null}) \Longrightarrow^p \#(\text{Null} \rightarrow \text{Null})} \text{ NORM-CAST}$$

Now we can use R-AppNorm to turn the origin application into

$$(t : \#(\text{Null} \rightarrow \text{Null}) \Longrightarrow^p \#(\text{Null} \rightarrow \text{Null})) \text{ null}$$

We can then proceed the evaluation using R-AppCast.

- **R-SafeAppNull** is simple: if we are evaluating a safe application $\text{app}(v, u, u')$ and the underlying function v is **null**, then the entire term reduces to u' (the default value).
- Finally, **R-SafeAppNorm** handles the remaining case. We have a safe application $\text{app}(v, u, u')$ like before, but this time we know that v is an abstraction (via $\mathbf{abs}(v)$). What we would like to do is to turn the safe application into a regular one: $\text{app}(v, u, u') \mapsto v u$. However, this can lead to the term getting stuck, if v is a cast to a safe nullable function (a $?$ function). The problem is that safe nullable functions are not supposed to appear in regular applications. The solution is to normalize v to v' . Since v' is guaranteed to have a regular function type after normalization, we can take the step $\text{app}(v, u, u') \mapsto v' u$, and then follow up with R-AppCast or R-App.

4.5.3 Blame Assignment

The blame assignment relation is responsible for determining which cast in a value is responsible for a nullability error. Once the responsible cast has been identified, blame assignment also determines whether the blame is *positive* (blame the *cast*) or *negative* (blame the context). The notation for blame assignment is $v \uparrow p$, and indicates that if the value v , containing one or more casts, leads to a failure (because $\mathbf{null}(v)$ holds and v was used in the function position of an application), then we will blame label p .

The rules for blame assignment are shown in Figure 5. There are two kinds of rules, based on what they do with the outermost cast: those that *discard* the outermost cast, and those that use the outermost cast to assign blame. Both kinds are described below.

Rules that discard the outermost cast:

- **B-NonNullable** handles the cast where the outermost cast has the form $v' : \#(S \rightarrow T) \Longrightarrow^p U$; that is, the source type is a *non-nullable* function type. Intuitively, we do not want to assign blame to either p or \bar{p} , because the *source* type in the cast promised that the underlying value is *non-null*, but the value being cast *is* in fact **null**. That is, there must be another “risky” cast that is part of v' that should be blamed. For example,

Reduction

$$\boxed{s \mapsto r}$$

$$E[(\lambda(x: T).s) v] \mapsto E[[v/x]s] \text{ (R-APP)}$$

$$\frac{\mathbf{null}(v) \quad v \uparrow p}{E[v u] \mapsto \uparrow p} \text{ (R-APPFAIL)}$$

$$\frac{\mathbf{abs}(v) \quad v \gg v'}{E[v u] \mapsto E[v' u]} \text{ (R-APPNORM)}$$

$$\frac{\mathbf{null}(v)}{E[\mathbf{app}(v, s, t)] \mapsto E[t]} \text{ (R-SAFEAPPNULL)}$$

$$\frac{\mathbf{abs}(v) \quad v \gg v'}{E[\mathbf{app}(v, s, t)] \mapsto E[v' s]} \text{ (R-SAFEAPPNORM)}$$

$$\frac{\mathbf{abs}(v)}{E[(v : \#(S_1 \rightarrow S_2) \Rightarrow^p \#(T_1 \rightarrow T_2)) u] \mapsto E[(v (u : T_1 \Rightarrow^{\bar{p}} S_1) : S_2 \Rightarrow^p T_2)]} \text{ (R-APPCAST)}$$

Evaluation contexts
 $E ::=$
 \square
 $E s$
 $v E$
 $\mathbf{app}(E, s, t)$
 $E : S \Rightarrow^p T$
Blame assignment

$$\boxed{v \uparrow p}$$

$$\frac{(v : \mathbf{Null} \Rightarrow^p!(S \rightarrow T)) \uparrow \bar{p} \text{ (B-NULL)}}{v \uparrow p'} \text{ (B-UNSAFE!)}$$

$$\frac{v \uparrow p'}{(v : \#(S \rightarrow T) \Rightarrow^p U) \uparrow p'} \text{ (B-NONNULLABLE)} \quad (v : ?(S \rightarrow T) \Rightarrow^p!(S' \rightarrow T')) \uparrow \bar{p} \text{ (B-SAFE!)}$$

$$\frac{\alpha \in \{?, !\}}{(v : \alpha (S \rightarrow T) \Rightarrow^p \#(S' \rightarrow T')) \uparrow p} \text{ (B-NULLABLE\#)}$$

Normalization

$$\boxed{v \gg u}$$

$$\lambda(x: T).s \gg \lambda(x: T).s \text{ (NORM-ABS)}$$

$$\frac{v \gg u \quad \alpha, \beta \in \{\#, ?, !\}}{v : \alpha (S_1 \rightarrow S_2) \Rightarrow^p \beta (T_1 \rightarrow T_2) \gg u : \#(S_1 \rightarrow S_2) \Rightarrow^p \#(T_1 \rightarrow T_2)} \text{ (NORM-CAST)}$$

■ **Figure 5** Reduction rules of $\lambda_{\mathbf{null}}$, along with blame assignment and normalization relations.

consider the cast $((\text{Null} \Rightarrow^r ?) \Rightarrow^q \#) \Rightarrow^p !$, where we have written only the top level “modalities” of the function types. In this cast, a `null` value that starts as having type `Null` is cast first to a safe nullable function, then to a non-nullable function, and finally to an unsafe nullable function. Blame assignment models the intuition that the second cast (from `?` to `#`) is the unsafe one, and so should be blamed. Because the destination type in that second cast is a `#`, we blame the term (i.e. blame q).

- **B-Unsafe!** is similar to the previous case: when confronted with a cast $v' : S \Longrightarrow^p T$ where *both* S and T are `!` types, then we “recurse” on v' to find the guilty cast. The reason is that the last cast did not change the kind of function type, so whatever went wrong must have happened earlier. For example, suppose the outermost cast is $!(\text{Null} \rightarrow \text{Null}) \Rightarrow^p !(\text{Null} \rightarrow \text{Null})$. This cast leaves the type unchanged, so it should never be blamed for a failure.

Notice that the equivalent rule for `#` types is subsumed by `B-NonNullable`. `?` types do not need an equivalent rule, because a cast of the form $v : S \Longrightarrow^p ?$ cannot fail.

Rules that assign blame based on the outermost cast:

- **B-Null** handles the case where we cast `Null` to an unsafe function type. In this case, we blame the context, because the target type is a `!`.
- **B-Nullable#** casts some kind of nullable function (either a `?` or a `!`) to a non-nullable function. In this case, we want to blame the term, because the context was promised a non-nullable value that nevertheless ended up being `null`.
- **B-Unsafe!** handles casts of the form $? \Rightarrow^p !$. In this case, we blame \bar{p} , because the context should know that the value is potentially `null`.

4.6 Metatheory of λ_{null}

In developing the metatheory, we closely followed the syntactic approach taken in Wadler and Findler [27]. All the results in this section have been verified using the Coq proof assistant.

4.6.1 Safety Lemmas

The first step is establishing that evaluation of well-typed λ_{null} terms does not get stuck. We do this by proving the classic progress and preservation lemmas due to Wright and Felleisen [29]. First, we need an auxiliary lemma that says that normalization preserves well-typedness.

► **Lemma 4** (Soundness of normalization). *Let $\alpha \in \{\#, ?, !\}$, $\Gamma \vdash v : \alpha (S \rightarrow T)$ and $v \gg v'$. Then $\Gamma \vdash v' : \#(S \rightarrow T)$.*

Then we can prove preservation.

► **Lemma 5** (Preservation). *Let $\Gamma \vdash t : T$ and suppose that $t \mapsto r$. Then either*

- $r = \uparrow p$, for some blame label p , or
- $r = t'$ for some term t' , and $\Gamma \vdash t' : T$

Notice that, because of unsafe casts like $\text{null} : \text{Null} \Longrightarrow^p !(S \rightarrow T)$, taking an evaluation step might lead to an error $\uparrow p$.

Before showing progress, we need a lemma that says that non-nullable values typed with a function type can be normalized.

► **Lemma 6** (Completeness of normalization). *Let $\alpha \in \{\#, ?, !\}$, $\Gamma \vdash v : \alpha (S \rightarrow T)$ and suppose that $\text{abs}(v)$ holds. Then there exists a value v' such that $v \gg v'$.*

$S <:^+ T$	$S <:^- T$
Null $<:^+$ Null (PS-NULLEFL)	Null $<:^-$ Null (NS-NULLEFL)
$\frac{\alpha \in \{?, !\}}{\text{Null } <:^+ \alpha (S \rightarrow T)} \text{ (PS-NULL)}$	Null $<:^-$ $?(S \rightarrow T)$ (NS-NULL)
$\frac{S' <:^- S \quad T <:^+ T' \quad \alpha \in \{\#, ?, !\}}{\#(S \rightarrow T) <:^+ \alpha (S' \rightarrow T')} \text{ (PS-ARROW\#)}$	$\frac{S' <:^+ S \quad T <:^- T' \quad \alpha \in \{\#, ?, !\}}{\#(S \rightarrow T) <:^- \alpha (S' \rightarrow T')} \text{ (NS-ARROW\#)}$
$\frac{S' <:^- S \quad T <:^+ T' \quad \alpha, \beta \in \{?, !\}}{\alpha (S \rightarrow T) <:^+ \beta (S' \rightarrow T')} \text{ (PS-ARROWNULLABLE)}$	$\frac{S' <:^+ S \quad T <:^- T' \quad \alpha \in \{\#, ?, !\}}{!(S \rightarrow T) <:^- \alpha (S' \rightarrow T')} \text{ (NS-ARROW!)}$
	$\frac{S' <:^+ S \quad T <:^- T' \quad \alpha \in \{\#, ?\}}{?(S \rightarrow T) <:^- \alpha (S' \rightarrow T')} \text{ (NS-ARROW?)}$

■ **Figure 6** Positive and negative subtyping.

This lemma is necessary because if we are ever evaluating a well-typed safe application (e.g. $\text{app}(v, u, u')$) where the function value (v) is known to be non-nullable, then we need to be able to turn the safe application into a regular application ($v u$) using R-SafeAppNorm .

We also need a weakening lemma.

► **Lemma 7** (Weakening). *Let $\Gamma \vdash t : T$ and $x \notin \text{dom}(\Gamma)$. Then $\Gamma, x : U \vdash t : T$ for any type U .*

We can then show progress.

► **Lemma 8** (Progress). *Let $\vdash t : T$. Then either*

- *t is a value*
- *$t \mapsto \uparrow p$, for some blame label p*
- *$t \mapsto t'$, for some term t'*

4.6.2 Blame Lemmas

The progress and preservation lemmas do not tell us as much as they usually do, because of the possibility of errors. It would then be nice to rule out errors in some cases. Examining the evaluation rules, we can notice that errors occur due to casts: specifically, because we sometimes cast a `null` value to a function type, which we later try to apply.

Inspecting the rules for blame assignment shows that casts to $!(T \rightarrow U)$ can lead to *negative* blame, and casts to $\#(T \rightarrow U)$ can lead to *positive* blame. We can then define two relations: *positive subtyping* ($T <:^+ U$) and *negative subtyping* ($T <:^- U$), that identify which casts *cannot* lead to positive and negative blame, respectively. The subtyping rules, adapted from Wadler and Findler [27], are shown in Figure 6.

► **Example 9.** Since the type system ensures that $?(S \rightarrow T)$ functions are only ever applied through safe casts, we would hope that the cast $\text{null} : \text{Null} \Rightarrow^p?(S \rightarrow T)$ will not fail with *either* blame $\uparrow p$ or $\uparrow \bar{p}$. Therefore we have both $\text{Null} <:^+ ?(S \rightarrow T)$ and $\text{Null} <:^- ?(S \rightarrow T)$.

$$\begin{array}{c}
\boxed{t \text{ safe for } p} \\
x \text{ safe for } p \quad (\text{SF-VAR}) \qquad \frac{S <:^+ T \quad s \text{ safe for } p}{s : S \Longrightarrow^p T \text{ safe for } p} (\text{SF-CASTPOS}) \\
\text{null safe for } p \quad (\text{SF-NULL}) \\
\frac{s \text{ safe for } p}{\lambda(x:T).s \text{ safe for } p} (\text{SF-ABS}) \qquad \frac{S <:^- T \quad s \text{ safe for } p}{s : S \Longrightarrow^{\bar{p}} T \text{ safe for } p} (\text{SF-CASTNEG}) \\
\frac{s \text{ safe for } p \quad t \text{ safe for } p}{s t \text{ safe for } p} (\text{SF-APP}) \\
\frac{f \text{ safe for } p \quad s \text{ safe for } p \quad t \text{ safe for } p}{\text{app}(f, s, t) \text{ safe for } p} (\text{SF-SAFEAPP}) \qquad \frac{s \text{ safe for } p \quad q \neq p \quad q \neq \bar{p}}{s : S \Longrightarrow^q T \text{ safe for } p} (\text{SF-CASTDIFF})
\end{array}$$

■ **Figure 7** Safe for relation.

► **Example 10.** Since a cast $\text{null} : \text{Null} \Longrightarrow^p!(S \rightarrow T)$ can fail with blame \bar{p} , we have $\text{Null} <:^+ !(S \rightarrow T)$, but not $\text{Null} <:^- !(S \rightarrow T)$.

► **Lemma 11** (Positive and negative subtyping are reflexive). *Let T be an arbitrary type. Then $T <:^+ T$ and $T <:^- T$.*

► **Lemma 12** (Subtyping implies compatibility). *Let S and T be types. Then*

- $S <:^+ T \Longrightarrow S \rightsquigarrow T$
- $S <:^- T \Longrightarrow S \rightsquigarrow T$

Lemma 12 implies that if S is a (positive or negative) subtype of T , then we can cast S to T (which requires compatibility).

The next step is to lift positive and negative subtyping to work on terms. The *safe for* relation, again adapted from Wadler and Findler [27] and shown in Figure 7, accomplishes this. We say that a term t is *safe for* a blame label p , written t **safe for** p , if evaluating t cannot lead to an error with blame p . That is, evaluating t either diverges, results in a value, or results in an error with blame different from p . We formalize this fact as a theorem below.

Most of the rules in the **safe for** relation just involve structural recursion on the subterms of a term. The connection with subtyping appears in SF-CastPos and SF-CastNeg. For example, to conclude that $(s : S \Longrightarrow^p T)$ **safe for** p , we require that s **safe for** p and $S <:^+ T$.

The following lemmas say that **safe for** is preserved by normalization and substitution.

► **Lemma 13** (Normalization preserves **safe for**). *Let v be a value such that v **safe for** p and suppose that $v \gg v'$. Then v' **safe for** p .*

► **Lemma 14** (Substitution preserves **safe for**). *Let t and t' be terms such that t **safe for** p and t' **safe for** p . Then $[t'/x]t$ **safe for** p .*

We now arrive at the main results in this section, the progress and preservation theorems for safe terms.

► **Theorem 15** (Preservation of safe terms). *Let $\Gamma \vdash t : T$ and t **safe for** p . Now suppose that t steps to a term t' (that is, taking an evaluation step from t is possible and does not result in an error). Then t' **safe for** p .*

- **Theorem 16** (Progress of safe terms). *Let $\vdash t : T$ and t **safe for p** . Then either*
- t is a value
 - $t \mapsto \uparrow p'$, for some blame label $p' \neq p$.
 - $t \mapsto t'$, for some term t'

Notice that this theorem does not preclude the term from stepping to an error, but it does say that the error *will not have blame label p* . This is a stronger guarantee than what we get from Lemma 8 (Progress), which placed no restrictions on the blame label p' when $t \mapsto \uparrow p'$.

Here are a few implications of the theorems above:

- A term without casts cannot fail. This is because a term can only fail with some blame label p , and a term without casts is necessarily **safe for p** .
- Casts that turn a “Java” type like $!(\text{Null} \rightarrow \text{Null}) \rightarrow \text{Null}$ into the corresponding “Scala” type $?(\text{Null} \rightarrow \text{Null}) \rightarrow \text{Null}$ via “nullification” can only fail with positive blame, because of negative subtyping.
- Conversely, casts that turn a “Scala” type like $\#(\text{Null} \rightarrow \text{Null}) \rightarrow \text{Null}$ into the corresponding “Java” type $!(\text{Null} \rightarrow \text{Null}) \rightarrow \text{Null}$ via erasure can only fail with negative blame, because of positive subtyping.

The last two claims form the bases for our model of language interoperability, described in the next section.

5 A Calculus for Null Interoperability

The λ_{null} calculus is very flexible in that it allows us to freely mix in implicitly nullable terms with explicitly nullable terms. On the other hand, it is perhaps *too flexible*. In the real world, when a language where `null` is explicit interoperates with a language where `null` is implicit, the separation between terms from both languages is very clear (it is usually enforced at a file or module boundary). For example, in the Java and Scala case, the Scala typechecker will *only* allow *explicit* nulls, while the Java typechecker *only* allows *implicit* nulls. To more faithfully model this kind of language interoperability, this section introduces a slight modification of λ_{null} called λ_{null}^s (“stratified lambda null”).

5.1 Terms and Types of λ_{null}^s

The terms and types of λ_{null}^s are shown in Figure 8. The main difference with respect to λ_{null} is that terms and types are stratified into the world of explicit nulls (subscript e) and the world of implicit nulls (subscript i). Notice that the grammar for types in the “explicit sublanguage” only allows for non-nullable functions ($\#(S \rightarrow T)$) and *safe* nullable functions ($?(S \rightarrow T)$). Similarly, the implicit sublanguage only has unsafe nullable functions ($!(S \rightarrow T)$). The only new terms are `imports`, which in the explicit sublanguage have syntax

$$\text{import}_e x : T_e = (t_i : T_i) \text{ in } t_e$$

Informally, an import term is similar to a let-binding: it binds x as having type T_e in the body t_e . *However*, the term that x is bound to, t_i , comes from the *implicit* sublanguage: it is a t_i and not a t_e . Furthermore, t_i is expected to have type T_i . Dually, the implicit sublanguage has an import term that binds x to an element of t_e , as opposed to a t_i :

$$\text{import}_i x : T_i = (t_e : T_e) \text{ in } t_i$$

$t ::=$	Terms		
t_e	terms with <i>explicit</i> nulls		
t_i	terms with <i>implicit</i> nulls		
$f_e, s_e, t_e ::=$	Explicit terms	$f_i, s_i, t_i ::=$	Implicit terms
x	variable	x	variable
<code>null</code>	null literal	<code>null</code>	null literal
$\lambda(x : T_e).s_e$	abstraction	$\lambda(x : T_i).(s_i : S_i)$	abstraction
$s_e t_e$	application	$s_i t_i$	application
<code>app(f_e, s_e, t_e)</code>	safe application	<code>app(f_i, s_i, t_i)</code>	safe application
$s_e : S_e \Longrightarrow T_e$	cast	$s_i : S_i \Longrightarrow^p T_i$	cast
<code>import_e $x : T_e = (t_i : T_i)$ in t_e</code>	import	<code>import_i $x : T_i = (t_e : T_e)$ in t_i</code>	import
$S_e, T_e ::=$	Explicit types	$S_i, T_i ::=$	Implicit types
<code>Null</code>	null	<code>Null</code>	null
$\#(S_e \rightarrow T_e)$	presumed non-nullable function	$!(S_i \rightarrow T_i)$	unsafe nullable function
$?(S_e \rightarrow T_e)$	safe nullable function		

■ **Figure 8** Terms and types of λ_{null}^s . Differences with λ_{null} are highlighted.

Imports allow us to link the world of explicit nulls with the world of implicit nulls, in much the same way as Scala’s `import` statements allow us to use Java libraries from Scala code (similarly, Java’s `import` statements allow us to use Scala libraries from Java code).

Casts in the explicit sublanguage do not have blame labels. This is because the type system will force all such casts to be *upcasts*: i.e. casts that respect subtyping. We will see that this means that “internal” casts within the explicit sublanguage will never be blamed for failures. Relatedly, notice that λ_{null}^s , unlike e.g. Scala, has no subsumption rule. We opted for casts instead of subsumption to keep λ_{null}^s close to λ_{null} . Subsumptions and casts are similarly expressive: one can think of subsumption as casts automatically introduced by the type checker.

Finally, abstractions in the implicit sublanguage, written $\lambda(x : T_i).(s : S_i)$, are annotated with their return type S_i . This is not strictly necessary, but it simplifies the presentation of desugaring in Section 5.3.

5.2 Typing λ_{null}^s

The typing rules for λ_{null}^s are shown in Figure 9. These rules are almost verbatim copies of the typing rules for λ_{null} (and the compatibility relation is reused from Figure 2). The two new rules handle imports:

- TE-Import handles the case where an implicitly nullable term is used from the world of explicit nulls. To type `importe $x : T_e = (t_i : T_i)$ in t_e` , we first type t_e in the context $\Gamma, x : T_e$, obtaining a type S_e . This will be the type of the entire term. The interesting twist comes next: the term t_i is typed with the \vdash_i relation in an *empty* context, so that $\emptyset \vdash_i t_i : T_i$. Finally, we need to somehow check that the type T_i determined by the \vdash_i relation and the type T_e expected by the \vdash_e relation are “in agreement”. This is done by the *nullification* relation, whose judgment is written $T_i \hookrightarrow_N T_e$, and is shown in Figure 10.
- TI-Import handles the opposite case, where a term from the world of explicit nulls is used in an implicitly nullable term. Here we use the “dual” of nullification: the *erasure* relation, written $T_e \hookrightarrow_E T_i$. Erasure is also shown in Figure 10.

► **Remark 17.** In designing TE-Import and TI-import, we have to decide under *which context* we will type the “embedded” term that comes from the foreign sublanguage. For simplicity, we have chosen to do the typechecking under the *empty* context. This prevents λ_{null}^s from modelling circular dependencies between terms of different languages, but otherwise seems not unduly restrictive.

Nullification and erasure, shown in Figure 10, are binary relations on types. They are inspired by how Java and Scala interoperate; specifically, the types of Java terms are “nullified” before being used by Scala code, and the types of Scala terms are “erased” before being used by Java code. Of course, the real-world nullification and erasure are more complicated than the simple relations presented here, but we believe the formalization in this section does capture the essence of how these relations affect nullability of types; namely, nullification conservatively assumes that every component of a Java type is nullable, while erasure eliminates the distinction between nullable and non-nullable types in the \vdash_e type system.

Notice that the typing rules for casts are now different in the explicit and implicit sublanguages. In the implicit sublanguage, like in λ_{null} , to type the cast $s_i : S_i \Longrightarrow^P T_i$, we require that S_i be *compatible* with T_i ($S_i \rightsquigarrow T_i$). By contrast, when typing casts in the explicit sublanguage, e.g. $s_e : S_e \Longrightarrow T_e$, we check that S_e can be *upcasted* to T_e , written $S_e <:_e T_e$. The upcasting is defined by the *explicit subtyping* relation, given in Figure 11. Explicit subtyping is defined just like we would define a regular subtyping relation, that is, it implies *substitutability* [17]. For example, we have the judgment $\#(S \rightarrow T) <:_e ?(S \rightarrow T)$, which is akin to the Scala judgment `String <: String|Null`.

Crucially, we can show that explicit subtyping implies *both* positive and negative subtyping.

► **Lemma 18.** $S <:_e T$ implies $S <:^+ T$ and $S <:^- T$.

This is useful, because it hints that casts that rely on explicit subtyping will never be blamed for failures.

5.3 Desugaring λ_{null}^s to λ_{null}

The last step is to give meaning to λ_{null}^s terms. We could repeat the approach followed for λ_{null} using operational semantics, but instead we will do something different. We will *desugar* λ_{null}^s terms and types to λ_{null} terms and types, respectively. This is useful, because in Section 4.6 we proved many results about λ_{null} terms, and we would like to re-use these results to reason about λ_{null}^s as well.

We will do the desugaring using a pair of functions $(\mathcal{D}_e, \mathcal{D}_i)$. \mathcal{D}_e is a function that sends λ_{null}^s terms from the explicit sublanguage to λ_{null} terms. Similarly, \mathcal{D}_i is a function that maps λ_{null}^s terms from the implicit sublanguage to λ_{null} terms. Both functions are shown in Figure 12.

The first thing to notice is that we do not actually need to desugar *types*. This is because λ_{null}^s types (from both sublanguages) are *also* λ_{null} types.

When it comes to terms, most cases in Figure 12 are handled by straightforward structural recursion on the term. There are only four interesting cases:

- **(DE-Cast)** Casts in the explicit sublanguage do not have blame labels, but casts in λ_{null} must always have labels. When we desugar explicit casts, we tag them with the same (“compiler-generated”) label \mathcal{E}_{int} . Later, we show that these casts are never blamed for failures (neither positively nor negatively).
- **(DI-Abs)** An abstraction $\lambda(x : S_i).(s_i : T_i)$ from the implicit sublanguage is typed as $!(S_i \rightarrow T_i)$ (Figure 9). However, the corresponding lambda in λ_{null} , $\lambda(x : S_i).\mathcal{D}_i(s_i)$, will have type $\#(S_i \rightarrow T_i)$. So that the metatheory in Section 5.4 works out, we need

$\Gamma \vdash_e t_e : T_e$	$\Gamma \vdash_i t_i : T_i$
$\frac{\Gamma(x) = T_e}{\Gamma \vdash_e x : T_e} \quad (\text{TE-VAR})$	$\frac{\Gamma(x) = T_i}{\Gamma \vdash_i x : T_i} \quad (\text{TI-VAR})$
$\Gamma \vdash_e \text{null} : \text{Null} \quad (\text{TE-NULL})$	$\Gamma \vdash \text{null} : \text{Null} \quad (\text{TI-NULL})$
$\frac{\Gamma, x : S_e \vdash_e s_e : T_e}{\Gamma \vdash_e \lambda(x : S_e).s_e : \#(S_e \rightarrow T_e)} \quad (\text{TE-ABS})$	$\frac{\Gamma, x : S_i \vdash_i s_i : T_i}{\Gamma \vdash_i \lambda(x : S_i).(s_i : T_i) : \!(S_i \rightarrow T_i)} \quad (\text{TI-ABS})$
$\frac{\Gamma \vdash_e s_e : \#(S_e \rightarrow T_e) \quad \Gamma \vdash_e t_e : S_e}{\Gamma \vdash_e s_e t_e : T_e} \quad (\text{TE-APP})$	$\frac{\Gamma \vdash_i s_i : \!(S_i \rightarrow T_i) \quad \Gamma \vdash_i t_i : S_i}{\Gamma \vdash_i s_i t_i : T_i} \quad (\text{TI-APP})$
$\frac{\Gamma \vdash_e f_e : \!(S_e \rightarrow T_e) \quad \Gamma \vdash_e s_e : S}{\Gamma \vdash_e \text{app}(f_e, s_e, t_e) : T_e} \quad (\text{TE-SAFEAPP})$	$\frac{\Gamma \vdash_i f_i : \!(S_i \rightarrow T_i) \quad \Gamma \vdash_i s_i : S_i}{\Gamma \vdash_i \text{app}(f_i, s_i, t_i) : T_i} \quad (\text{TI-SAFEAPP})$
$\frac{\Gamma \vdash_e s_e : S_e \quad S_e <:_e T_e}{\Gamma \vdash_e (s_e : S_e \Longrightarrow T_e) : T_e} \quad (\text{TE-CAST})$	$\frac{\Gamma \vdash_i s : S_i \quad S_i \rightsquigarrow T_i}{\Gamma \vdash_i (s_i : S_i \Longrightarrow^p T_i) : T_i} \quad (\text{TI-CAST})$
$\frac{\Gamma, x : T_e \vdash_e t_e : S_e \quad \emptyset \vdash_i t_i : T_i \quad T_i \hookrightarrow_N T_e}{\Gamma \vdash_e \text{import}_e x : T_e = (t_i : T_i) \text{ in } t_e : S_e} \quad (\text{TE-IMPORT})$	$\frac{\Gamma, x : T_i \vdash_i t_i : S_i \quad \emptyset \vdash_e t_e : T_e \quad T_e \hookrightarrow_E T_i}{\Gamma \vdash_i \text{import}_i x : T_i = (t_e : T_e) \text{ in } t_i : S_i} \quad (\text{TI-IMPORT})$

■ **Figure 9** Typing rules of λ_{null}^s .

$$\begin{array}{c}
\boxed{T_i \hookrightarrow_N T_e} \qquad \boxed{T_e \hookrightarrow_E T_i} \\
\text{Null} \hookrightarrow_N \text{Null} \quad (\text{N-NULL}) \qquad \text{Null} \hookrightarrow_E \text{Null} \quad (\text{E-NULL}) \\
\\
\frac{S_i \hookrightarrow_N S_e \quad T_i \hookrightarrow_N T_e}{!(S_i \rightarrow T_i) \hookrightarrow_N ?(S_e \rightarrow T_e)} (\text{N-ARROW!}) \qquad \frac{S_e \hookrightarrow_E S_i \quad T_e \hookrightarrow_E T_i}{?(S_e \rightarrow T_e) \hookrightarrow_E !(S_i \rightarrow T_i)} (\text{E-ARROW?}) \\
\\
\frac{S_e \hookrightarrow_E S_i \quad T_e \hookrightarrow_E T_i}{\#(S_e \rightarrow T_e) \hookrightarrow_E !(S_i \rightarrow T_i)} (\text{E-ARROW\#})
\end{array}$$

■ **Figure 10** Nullification and erasure relations.

$$\begin{array}{c}
\text{Null} <:_e \text{Null} \qquad (\text{ES-NULLREFL}) \\
\text{Null} <:_e ?(S \rightarrow T) \qquad (\text{ES-NULL?}) \\
\frac{S' <:_e S \quad T <:_e T'}{\#(S \rightarrow T) <:_e \#(S' \rightarrow T')} \qquad (\text{ES-ARROW\#}) \\
\frac{S' <:_e S \quad T <:_e T'}{\#(S \rightarrow T) <:_e ?(S' \rightarrow T')} \qquad (\text{ES-ARROW?}) \\
\frac{S' <:_e S \quad T <:_e T'}{?(S \rightarrow T) <:_e ?(S' \rightarrow T')} \qquad (\text{ES-SAFE})
\end{array}$$

■ **Figure 11** Explicit subtyping (upcast) relation.

the types to match; hence the cast. This is another instance of a blame label being automatically inserted by desugaring. We will use the blame label \mathcal{I}_{int} : the \mathcal{I} stands for *implicit*, indicating that the term being cast is from the implicit sublanguage. The $_{\text{int}}$ subscript indicates that it is an *internal* cast; that is, it does not occur at the boundary between the implicit and explicit sublanguages. To do the cast, we need the return type T_i of the function: this is why abstractions in the implicit sublanguage contain type annotations for the return type.

- **(DE-Import)** This handles the case where we import a term from the implicit world into the explicit world. There are two desugarings that happen in this rule. The first is a standard desugaring that turns the import (effectively, a let binding) into a lambda abstraction that is immediately applied. In this way, we do not need to add let bindings to λ_{null} . The second desugaring is the insertion of a cast that “guards” the transformation of the original implicit type T_i into the explicit type T_e . The cast has blame label \mathcal{I} to indicate that the term being cast is from the implicit world (conversely, we could say that the *context* using the term is from the explicit world).
- **(DI-Import)** We also need a dual rule for importing a term from the *explicit* world into the implicit world. This rule does the same as (DE-Import), except that the cast now goes in the opposite direction: from T_e to T_i . The cast is labelled with blame \mathcal{E} , indicating that the term being cast comes from the explicit sublanguage.

$$\boxed{\mathcal{D}_e : s_e \longrightarrow s}$$

$$\begin{aligned} \mathcal{D}_e(x) &= x && \text{(DE-Var)} \\ \mathcal{D}_e(\mathbf{null}) &= \mathbf{null} && \text{(DE-Null)} \\ \mathcal{D}_e(\lambda(x : T_e).s_e) &= \lambda(x : T_e).\mathcal{D}_e(s_e) && \text{(DE-Abs)} \\ \mathcal{D}_e(s_e t_e) &= \mathcal{D}_e(s_e) \mathcal{D}_e(t_e) && \text{(DE-App)} \\ \mathcal{D}_e(\mathbf{app}(f_e, s_e, t_e)) &= \mathbf{app}(\mathcal{D}_e(f_e), \mathcal{D}_e(s_e), \mathcal{D}_e(t_e)) && \text{(DE-SafeApp)} \\ \mathcal{D}_e(s_e : S_e \Longrightarrow T_e) &= \mathcal{D}_e(s_e) : S_e \Longrightarrow^{\mathcal{E}_{\text{int}}} T_e && \text{(DE-Cast)} \\ \mathcal{D}_e(\mathbf{import}_e x_e : T_e = (t_i : T_i) \text{ in } t_e) &= (\lambda(x : T_e).\mathcal{D}_e(t_e)) (\mathcal{D}_i(t_i) : T_i \Longrightarrow^{\mathcal{I}} T_e) && \text{(DE-Import)} \end{aligned}$$

$$\boxed{\mathcal{D}_i : s_i \longrightarrow s}$$

$$\begin{aligned} \mathcal{D}_i(x) &= x && \text{(DI-Var)} \\ \mathcal{D}_i(\mathbf{null}) &= \mathbf{null} && \text{(DI-Null)} \\ \mathcal{D}_i(\lambda(x : S_i).(s_i : T_i)) &= (\lambda(x : S_i).\mathcal{D}_i(s_i)) : \#(S_i \rightarrow T_i) \Longrightarrow^{\mathcal{I}_{\text{int}}}!(S_i \rightarrow T_i) && \text{(DI-Abs)} \\ \mathcal{D}_i(s_i t_i) &= \mathcal{D}_i(s_i) \mathcal{D}_i(t_i) && \text{(DI-App)} \\ \mathcal{D}_i(\mathbf{app}(f_i, s_i, t_i)) &= \mathbf{app}(\mathcal{D}_i(f_i), \mathcal{D}_i(s_i), \mathcal{D}_i(t_i)) && \text{(DI-SafeApp)} \\ \mathcal{D}_i(s_i : S_i \Longrightarrow^p T_i) &= \mathcal{D}_i(s_i) : S_i \Longrightarrow^p T_i && \text{(DI-Cast)} \\ \mathcal{D}_i(\mathbf{import}_i x_i : T_i = (t_e : T_e) \text{ in } t_i) &= (\lambda(x : T_i).\mathcal{D}_i(t_i)) (\mathcal{D}_e(t_e) : T_e \Longrightarrow^{\mathcal{E}} T_i) && \text{(DI-Import)} \end{aligned}$$

■ **Figure 12** Desugaring $\lambda_{\mathbf{null}}^s$ terms to $\lambda_{\mathbf{null}}$ terms.

5.4 Metatheory of $\lambda_{\mathbf{null}}^s$

The following lemma shows that nullification implies negative subtyping, and erasure implies positive subtyping.

► **Lemma 19.** *Let S and T be types. Then $S \hookrightarrow_N T$ implies $S <:- T$ and $T <:+ S$, and $S \hookrightarrow_E T$ implies $S <:+ T$ and $T <:- S$.*

This is important because nullification is used to import implicit terms into the explicit world. The lemma shows that nullification implies negative subtyping, and casts where the arguments are negative subtypes never fail with *negative* blame. This means that if nullification-related casts fail, they do so by blaming the *term* being cast (which belongs to the implicit world), and never the context (which belongs to the explicit world). That is, the code with implicit nulls is at fault!

Dually, erasure is used to import explicit terms into the implicit world. Since erasure implies positive subtyping, then erasure-related casts can only fail with negative blame. That is, the *context* (which belongs to the implicit world) is at fault for erasure-related failures. Again, implicit nulls are to blame!

► **Theorem 20** (Desugaring preserves typing). *Let t_e and t_i be explicit and implicit terms from λ_{null}^s , respectively. Then*

- $\Gamma \vdash_e t_e : T_e \implies \Gamma \vdash \mathcal{D}_e(t_e) : T_e$, and
- $\Gamma \vdash_i t_i : T_i \implies \Gamma \vdash \mathcal{D}_i(t_i) : T_i$

► **Definition 21** (Set of user-written blame labels in a term). *We will denote the set of user-written blame labels in a term t of λ_{null}^s by $\mathbf{labels}(t)$. We do not give an explicit definition here, but $\mathbf{labels}(t)$ can be defined inductively on the structure of terms. Notice that user-written blame labels can only come from implicit casts $s_i : S_i \implies^P T_i$.*

The next theorem is our main result: it characterizes the failures that can occur while evaluating a (desugared) λ_{null}^s term. Specifically, it says that:

- Upcasts within the explicit world, which have blame \mathcal{E}_{int} , are *never blamed* for failures, neither positively nor negatively.
- Interop casts that result from importing an implicit term into an explicit term can only fail with *positive* blame, that is, they blame \mathcal{I} . This means the term being cast, which originated in the implicit sublanguage, is at fault.
- Interop casts that result from importing an explicit term into an implicit term can only fail with *negative* blame, that is, they blame $\bar{\mathcal{E}}$. If the blame is $\bar{\mathcal{E}}$, then the *context* surrounding the term being cast is at fault; in this case, the term being cast comes from the explicit sublanguage, so the context is in the implicit sublanguage.
- Internal casts tagged with \mathcal{I}_{int} , which result from desugaring $\lambda(x : S_i).(s_i : T_i)$ expressions, are *never blamed* for failures, neither positively nor negatively. That is, the desugaring does not introduce faulty casts.
- User-written casts ($s_i : S_i \implies^P T_i$) within the implicit sublanguage can still be blamed, but that is expected because some of those casts are indeed unsafe.

► **Theorem 22** (Explicitly nullable programs can't be blamed). *Let t be a term of λ_{null}^s . Suppose that $\{\mathcal{I}, \bar{\mathcal{I}}, \mathcal{I}_{\text{int}}, \bar{\mathcal{I}}_{\text{int}}, \mathcal{E}, \bar{\mathcal{E}}, \mathcal{E}_{\text{int}}, \bar{\mathcal{E}}_{\text{int}}\} \cap \mathbf{labels}(t) = \emptyset$. Further, suppose that t is well-typed under \vdash_e or \vdash_i and a context Γ . Then*

- If $t = t_e$, then $\mathcal{D}_e(t_e)$ **safe for** $\{\mathcal{E}_{\text{int}}, \bar{\mathcal{E}}_{\text{int}}, \bar{\mathcal{I}}, \mathcal{E}, \mathcal{I}_{\text{int}}, \bar{\mathcal{I}}_{\text{int}}\}$.⁸
- If $t = t_i$, then $\mathcal{D}_i(t_i)$ **safe for** $\{\mathcal{E}_{\text{int}}, \bar{\mathcal{E}}_{\text{int}}, \bar{\mathcal{I}}, \mathcal{E}, \mathcal{I}_{\text{int}}, \bar{\mathcal{I}}_{\text{int}}\}$.

Just like a central result in gradual typing is that “well-typed programs can't be blamed” [27], we can summarize our main result as *explicitly nullable programs can't be blamed*.

6 Coq Mechanization

All our results have been verified using the Coq theorem prover. The two main differences between the presentation of λ_{null} in this paper and in the Coq proofs are:

- The definition of evaluation in the Coq code does not use evaluation contexts, unlike Figure 5. Instead, we have explicit rules for propagating errors.
- The definition of terms in the Coq code uses a locally-nameless representation of terms [5].

In the mechanization of the proofs, we used the Ott [21] and LNgen [2] tools, which automate the generation of some useful auxiliary lemmas from a description of the language grammar. In total, the Coq code has 4657 lines of code, of which 1423 are manually-written proofs, while the rest are either library code or automatically-generated by Ott and LNgen.

⁸ The notation t **safe for** L , where L is a set of blame labels, indicates that t **safe for** l for every $l \in L$.

7 Related Work

The concept of blame comes from work on higher-order contracts by Findler and Felleisen [11]. The application of blame to gradual typing was pioneered by Tobin-Hochstadt and Felleisen [25], and Wadler and Findler [27]. We followed the latter closely when developing the operational semantics and safety proofs for λ_{null}^s . Our syntax for casts comes from Ahmed et al. [1]. Wadler [26] provides additional context on the use of blame for gradual typing.

The *gradual guarantee*, introduced by Siek et al. [23], is a property of gradually-typed languages that characterizes the behaviour of terms as type annotations are added or removed from a program. Roughly speaking, removing type annotations preserves program behaviour, while adding type annotations can lead only to certain classes of errors. In this way, languages that satisfy the gradual guarantee allow well-behaved migrations of untyped code into the typed world. Determining whether λ_{null}^s satisfies a property analogous to the gradual guarantee remains future work.

Linking types [19] solve the related (and more general) problem of ensuring that typing guarantees that hold in one or more source languages (e.g. Java and Scala) continue to hold, after compilation, in a target language (e.g. JVM bytecode), even in the presence of linking. However, linking types require that the source languages be augmented with additional types (the linking types), and that the target language be sufficiently expressive. In the case of `null` interoperability for Java and Scala, for example, this would mean adding a notion of nullable types both to Java (the source language) and JVM bytecode (the target language). This makes the `null` interoperability problem trivial, but would require considerable additional effort, when compared to our approach.

Multiple modern programming languages have types that are non-nullable by default. Examples include Kotlin [16], Swift [13], C# [6], and (recently) Scala [8]. In all of these, it is possible to recover nullability at the type level. For example, in Kotlin the type `String` is non-nullable, but `String?` is nullable. In Scala, nullability is expressed as a special case of *type unions*: `String|Null` represents nullable strings. Additionally, all of these languages also need to support some form of interoperability with a “less-precisely typed” language, where nullability remains implicit and is not tracked in the types. In the Kotlin and Scala case, the less-precisely typed language is Java; for Swift, it is Objective-C; and for C#, it is any language that compiles to the .NET runtime.

All of the languages above make pragmatic design decisions in their `null` interoperability. Specifically, their versions of type nullification trade off soundness for usability. For example, in Kotlin, a `String` type flowing from Java is translated as the *platform type* [14] `String!`, as opposed to `String?`. Platform types allow different kinds of unsound, yet convenient, behaviour. For example, we can select fields and methods on a platform type, or assign a platform type to the corresponding non-nullable type (e.g. assign a `String!` to a `String`). Naturally, these unsafe operations might fail at runtime. Similarly to platform types in Kotlin, Swift has *implicitly unwrapped optionals* and Scala has an `UncheckedNull` type (which has fewer soundness holes, but does not help as much with usability).

The design of λ_{null}^s was inspired by `null` interoperability in Scala and Kotlin. The main difference is that type nullification is “sound” in λ_{null}^s : that is, the unsafe nullable type $!(S \rightarrow T)$ is translated into the *safe* nullable type $?(S \rightarrow T)$. However, as we have seen, nullability errors remain, which motivates the use of blame to assign responsibility.

8 Conclusions

In this paper, we looked at the problem of characterizing the nullability errors that occur from two interoperating languages: one with explicit `nulls`, the other with implicit `nulls`. We showed how the concept of *blame* from gradual typing can be co-opted to provide such a characterization. Specifically, by making type casts explicit and labelling casts with blame labels, we are able to assign responsibility for runtime failures. To formally study the use of blame for tracking nullability errors, we introduced λ_{null} , a calculus where terms can be explicitly nullable or implicitly nullable. We showed that even though evaluation of λ_{null} terms can fail, such failures can be constrained if we restrict casts using positive and negative subtyping. Finally, we used λ_{null} as the basis for a higher-level calculus, λ_{null}^s , which more closely models language interoperability. Our main result is a theorem that says that *explicitly nullable programs can't be blamed* for null interoperability errors in λ_{null}^s .

References

- 1 Amal Ahmed, Robert Bruce Findler, Jeremy G. Siek, and Philip Wadler. Blame for all. 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 201–214. ACM, 2011.
- 2 Brian Aydemir and Stephanie Weirich. Lngen: Tool support for locally nameless representations. Technical Report MS-CIS-10-24, Computer and Information Science, University of Pennsylvania, 2010.
- 3 Subarno Banerjee, Lazaro Clapp, and Manu Sridharan. Nullaway: Practical type-based null safety for java. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 740–750. ACM, 2019.
- 4 Dan Brotherston, Werner Dietl, and Ondřej Lhoták. Granular: Gradual nullable types for java. In *Proceedings of the 26th International Conference on Compiler Construction*, pages 87–97. ACM, 2017.
- 5 Arthur Charguéraud. The locally nameless representation. *Journal of automated reasoning*, 49(3):363–408, 2012.
- 6 Microsoft Corporation. Nullable reference types. [Online; accessed 5-November-2019]. URL: <https://docs.microsoft.com/en-us/dotnet/csharp/nullable-references>.
- 7 Werner Dietl, Stephanie Dietzel, Michael D Ernst, Kivanç Muşlu, and Todd W Schiller. Building and using pluggable type-checkers. In *Proceedings of the 33rd International Conference on Software Engineering*, pages 681–690. ACM, 2011.
- 8 Doty Team. Explicit nulls. [Online; accessed 9-January-2020]. URL: <https://dotty.epfl.ch/docs/reference/other-new-features/explicit-nulls.html>.
- 9 Manuel Fähndrich and K. Rustan M. Leino. Declaring and checking non-null types in an object-oriented language. In Ron Crocker and Guy L. Steele Jr., editors, *Proceedings of the 2003 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications, OOPSLA 2003, October 26-30, 2003, Anaheim, CA, USA*, pages 302–312. ACM, 2003.
- 10 Manuel Fähndrich and Songtao Xia. Establishing object invariants with delayed types. In Richard P. Gabriel, David F. Bacon, Cristina Videira Lopes, and Guy L. Steele Jr., editors, *Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2007, October 21-25, 2007, Montreal, Quebec, Canada*, pages 337–350. ACM, 2007.

- 11 Robert Bruce Findler and Matthias Felleisen. Contracts for higher-order functions. In Mitchell Wand and Simon L. Peyton Jones, editors, *Proceedings of the Seventh ACM SIGPLAN International Conference on Functional Programming (ICFP '02), Pittsburgh, Pennsylvania, USA, October 4-6, 2002*, pages 48–59. ACM, 2002.
- 12 Arjun Guha, Claudiu Saftoiu, and Shriram Krishnamurthi. Typing local control and state using flow analysis. In *European Symposium on Programming*, pages 256–275. Springer, 2011.
- 13 Apple Inc. Swift language guide. [Online; accessed 5-November-2019]. URL: <https://docs.swift.org/swift-book/LanguageGuide/TheBasics.html>.
- 14 JetBrains. Calling Java code from Kotlin. [Online; accessed 9-January-2020]. URL: <https://kotlinlang.org/docs/reference/java-interop.html>.
- 15 JetBrains. Kotlin programming language. [Online; accessed 5-November-2019]. URL: <https://kotlinlang.org/>.
- 16 JetBrains. Null safety. [Online; accessed 5-November-2019]. URL: <https://kotlinlang.org/docs/reference/null-safety.html>.
- 17 Barbara Liskov and Jeannette M. Wing. A behavioral notion of subtyping. *ACM Trans. Program. Lang. Syst.*, 16(6):1811–1841, 1994.
- 18 Robin Milner. A theory of type polymorphism in programming. *J. Comput. Syst. Sci.*, 17(3):348–375, 1978.
- 19 Daniel Patterson and Amal Ahmed. Linking types for multi-language software: Have your cake and eat it too. In Benjamin S. Lerner, Rastislav Bodík, and Shriram Krishnamurthi, editors, *2nd Summit on Advances in Programming Languages, SNAPL 2017, May 7-10, 2017, Asilomar, CA, USA*, volume 71 of *LIPICs*, pages 12:1–12:15. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2017.
- 20 Xin Qi and Andrew C. Myers. Masked types for sound object initialization. In Zhong Shao and Benjamin C. Pierce, editors, *Proceedings of the 36th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2009, Savannah, GA, USA, January 21-23, 2009*, pages 53–65. ACM, 2009.
- 21 Peter Sewell, Francesco Zappa Nardelli, Scott Owens, Gilles Peskine, Thomas Ridge, Susmit Sarkar, and Rok Strnisa. Ott: Effective tool support for the working semanticist. *J. Funct. Program.*, 20(1):71–122, 2010.
- 22 Jeremy G Siek and Walid Taha. Gradual typing for functional languages. In *Scheme and Functional Programming Workshop*, volume 6, pages 81–92, 2006.
- 23 Jeremy G. Siek, Michael M. Vitousek, Matteo Cimini, and John Tang Boyland. Refined criteria for gradual typing. In Thomas Ball, Rastislav Bodík, Shriram Krishnamurthi, Benjamin S. Lerner, and Greg Morrisett, editors, *1st Summit on Advances in Programming Languages, SNAPL 2015, May 3-6, 2015, Asilomar, California, USA*, volume 32 of *LIPICs*, pages 274–293. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2015.
- 24 Alexander J. Summers and Peter Müller. Freedom before commitment: a lightweight type system for object initialisation. In Cristina Videira Lopes and Kathleen Fisher, editors, *Proceedings of the 26th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2011, part of SPLASH 2011, Portland, OR, USA, October 22 - 27, 2011*, pages 1013–1032. ACM, 2011.
- 25 Sam Tobin-Hochstadt and Matthias Felleisen. Interlanguage migration: from scripts to programs. In *OOPSLA Companion*, pages 964–974. ACM, 2006.
- 26 Philip Wadler. A complement to blame. In Thomas Ball, Rastislav Bodík, Shriram Krishnamurthi, Benjamin S. Lerner, and Greg Morrisett, editors, *1st Summit on Advances in Programming Languages, SNAPL 2015, May 3-6, 2015, Asilomar, California, USA*, volume 32 of *LIPICs*, pages 309–320. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2015.
- 27 Philip Wadler and Robert Bruce Findler. Well-typed programs can't be blamed. In Giuseppe Castagna, editor, *Programming Languages and Systems, 18th European Symposium on Programming, ESOP 2009, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009, York, UK, March 22-29, 2009. Proceedings*, volume 5502 of *Lecture Notes in Computer Science*, pages 1–16. Springer, 2009.

3:28 Blame for Null

- 28 Niklaus Wirth and C. A. R. Hoare. A contribution to the development of ALGOL. *Commun. ACM*, 9(6):413–432, 1966.
- 29 Andrew K. Wright and Matthias Felleisen. A syntactic approach to type soundness. *Inf. Comput.*, 115(1):38–94, 1994.
- 30 Yoav Zibin, David Cunningham, Igor Peshansky, and Vijay Saraswat. Object initialization in x10. In *European Conference on Object-Oriented Programming*, pages 207–231. Springer, 2012.

Static Race Detection and Mutex Safety and Liveness for Go Programs

Julia Gabet 

Imperial College London, United Kingdom
j.gabet18@imperial.ac.uk

Nobuko Yoshida 

Imperial College London, United Kingdom
n.yoshida@imperial.ac.uk

Abstract

Go is a popular concurrent programming language thanks to its ability to efficiently combine concurrency and systems programming. In Go programs, a number of concurrency bugs can be caused by a mixture of data races and communication problems. In this paper, we develop a theory based on behavioural types to statically detect data races and deadlocks in Go programs. We first specify lock safety/liveness and data race properties over a Go program model, using the happens-before relation defined in the Go memory model. We represent these properties of programs in a μ -calculus model of types, and validate them using type-level model-checking. We then extend the framework to account for Go's channels, and implement a static verification tool which can detect concurrency errors. This is, to the best of our knowledge, the first static verification framework of this kind for the Go language, uniformly analysing concurrency errors caused by a mix of shared memory accesses and asynchronous message-passing communications.

2012 ACM Subject Classification Software and its engineering → Concurrent programming languages; Software and its engineering → Model checking; Theory of computation → Process calculi

Keywords and phrases Go language, behavioural types, race detection, happens-before relation, safety, liveness

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2020.4

Related Version A full version of the paper is available at [13], <https://arxiv.org/abs/2004.12859>.

Supplementary Material ECOOP 2020 Artifact Evaluation approved artifact available at <https://doi.org/10.4230/DARTS.6.2.12>.

The source code for the tool presented in this paper and instructions to run it are available at [2, 1, 3].

Funding The work is partially supported by VeTSS, EPSRC EP/K011715/1, EP/K034413/1, EP/L00058X/1, EP/N027833/1, EP/N028201/1, EP/T006544/1 and EP/T014709/1.

Acknowledgements The authors want to thank Nicholas Ng for his initial collaboration on the project.

1 Introduction

Go is a concurrent programming language designed by Google for *programming at scale* [35]. Over the last few years, it has seen rapid growth and adoption: for instance in 2018, major developer surveys [12] show that StackOverflow placed Go in the top 5 most loved and the top 5 most wanted languages; and Github has reported in [14] that Go was the 7th fastest growing language.

One of the core pillars of Go is concurrent programming features, including the locking of shared memory for thread synchronisation, and the use of explicit message passing through channels, inspired by process calculi concurrency models [22, 31]. In practice, shared accesses



© Julia Gabet and Nobuko Yoshida;

licensed under Creative Commons License CC-BY

34th European Conference on Object-Oriented Programming (ECOOP 2020).

Editors: Robert Hirschfeld and Tobias Pape; Article No. 4; pp. 4:1–4:30

Leibniz International Proceedings in Informatics



LIPICs Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany



to memory using locking mechanisms are unavoidable, and could be accidental. It is also of note that both shared memory and message passing operations provide a substantial part of the concurrency features of Go, and are the ones that are more prone to misuse-induced bugs. These unsafe memory accesses may lead to *data races*, where programs silently enter an inconsistent execution state leading to hard-to-debug failures.

```

1 func main() {
2     var x int
3     m := new(sync.RWMutex)
4     go f(m, &x)
5     m.RLock() // acquire the lock for reading
6     x += 10   // write not protected by the lock
7     m.RUnlock() // release the read-lock
8     m.Lock() // acquire the lock for writing
9     fmt.Println("x is", x)
10    m.Unlock() // release the write-lock
11 }
12
13 func f(m *sync.RWMutex, ptr *int) {
14     m.RLock()
15     *ptr += 20 // write not protected by the lock
16     m.RUnlock()
17 }

```

■ **Figure 1** Go program with RWMutex (unsafe).

Figure 1 illustrates a Go program, which makes use of lock `m` to synchronise the `main` and `f` functions updating the content of variable `x`. On line 3, the statement `m := new(sync.RWMutex)` creates a new read-write lock `m`, called `RWMutex` in Go, used to guard memory accesses based on their status as readers or writers. The `RWMutex` object can then be passed around directly as on line 4, circumventing the issue that could arise if we copied the mutex structure instead. It can be locked for writing by calling its `Lock()` method, unlocked from writing handle with its `Unlock()` method, and locked and unlocked for reading with the `RLock()` and `RUnlock()` methods. Readers and writers are mutually exclusive, and writers are mutually exclusive to each other too (hence the name `Mutex`, for *mutual exclusion* lock), but an arbitrary number of readers can hold the lock at the same time. The `go` keyword in front of a function call on line 4 spawns a lightweight thread (called a *goroutine*) to execute the body of function `f`. The two parameters of function `f` – a `rwmutex` `m`, and an `int` pointer `ptr` – are shared between the caller and callee goroutines, `main` and `f`. Since concurrent access to the shared pointer `ptr` may introduce a data race, the developer tries to ensure serialised, mutually exclusive access to `ptr` in `f` and `x` in `main` by using read-locks. Using read-locks is unsafe in this case, allowing simultaneous write requests to `x` on lines 6 and 15, the program could then output “`x is 20`” with a bad scheduling, dropping the increase of 10 in the same thread as the print statement.

Figure 2 illustrates the same Go program, using the `RWMutex` feature correctly by putting writer sections of the code under writer locks. This alone prevents the data race seen in the first version of the program.

Go provides an optional *runtime* data race detector [48, 15] as a part of the Go compiler toolchain. The race detector is based on LLVM’s ThreadSanitizer [40, 45, 41] library, which detects races that manifest during execution. It can be enabled by building a program using the “`-race`” flag. During the program execution, the race detector creates up to four shadow words for every memory object to store historical accesses of the object. It compares every new access with the stored shadow word values to detect possible races. These runtime operations cause high overheads of the runtime detector (5–10 times overhead in memory usage and 2–20 times in execution time on average [15]), hence it is unrealistic to run it with

```

1 func main() {
2     var x int
3     m := new(sync.RWMutex)
4     go f(m, &x)
5     m.Lock() // acquire the lock for writing
6     x += 10 // protected by the lock
7     m.Unlock() // release the write-lock
8     m.RLock() // acquire the lock for reading
9     fmt.Println("x is", x)
10    m.RUnlock() // release the read-lock
11 }
12
13 func f(m *sync.RWMutex, ptr *int) {
14     m.Lock()
15     *ptr += 20 // protected by the lock
16     m.Unlock()
17 }

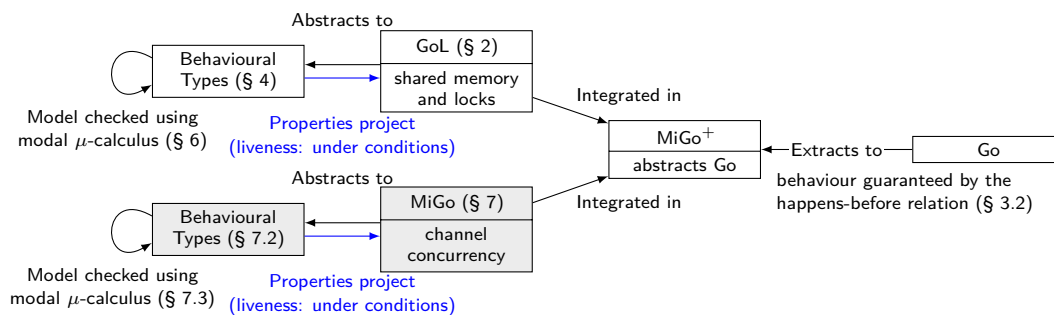
```

■ **Figure 2** Go program with RWMutex (safe).

race detection turned on in production code; and because of that, race detection relies on extensive testing or fuzzing techniques [47, 43]. Moreover, as reported in [46], the detector fails to find many non-blocking bugs as it cannot keep a sufficiently long enough history; and its semantics does not capture Go specific non-blocking bugs.

The Go memory model [16] defines the behaviour of memory access in Go as a *happens-before* relation by a *combination* of shared memory and channel communications. It is also reported in [46] that the most difficult bugs to detect are caused when synchronisation mechanisms are used together with message passing operations. For instance, Go can use message passing for sharing memory (channel-as-lock) or passing pointers through channels (pointer-through-channel), which might lead to a serious non-blocking bug, i.e. the program may continue to execute in unwanted and incorrect states or corrupt data in its computations [46], due to subtle interplays with buffered asynchronous communications.

These motivate us to *uniformly* model, statically analyse and detect concurrent non-blocking/blocking shared memory/channel-communications bugs in Go, using a formal model based on a process calculus [22, 31].



■ **Figure 3** Overview of this paper.

Contributions and Outline. Figure 3 outlines the relationship between the results presented in this paper. This work proposes a *uniform* model which handles first shared memory concurrency (§ 2), and then message-passing concurrency (§ 7) based on *concurrent behavioural types*, and presents the theory, design and implementation of a concurrent bug detector for Go. We formalise a happens-before relation and several key safety and liveness properties in the process calculus following the Go memory model [16] (§ 3). More specifically, in this

$ \begin{aligned} P, Q, R &::= \mu; P \mid (P \mid Q) \mid \mathbf{0} \mid (\nu u)P \\ &\mid \text{if } e \text{ then } P \text{ else } Q \\ &\mid X(\tilde{e}, \tilde{u}) \mid \text{new}(x : \sigma); P \\ &\mid \text{newl}(l); P \mid \text{newrw}(l); P \\ &\mid [x, \sigma :: v] \mid [l] \mid [l]^* \\ &\mid \langle l \rangle_i \mid \langle l \rangle_i^* \mid \langle l \rangle_i^\nabla \end{aligned} $	$ \begin{aligned} D &::= X(\tilde{x}) = P \\ \mathbf{P} &::= \{D_i\}_{i \in I} \text{ in } P \\ \mu &::= \tau \mid y \leftarrow \text{load}(x) \\ &\mid \text{store}(x, e) \mid \ell \\ \ell &::= \text{lock}(l) \mid \text{unlock}(l) \\ &\mid \text{rlock}(l) \mid \text{runlock}(l) \\ v &::= \mathbf{n} \mid \text{true} \mid \text{false} \mid x \\ e &::= v \mid \text{not}(e) \mid \text{succ}(e) \end{aligned} $
$ \begin{aligned} P \mid Q &\equiv Q \mid P & P \mid (Q \mid R) &\equiv (P \mid Q) \mid R & P \mid \mathbf{0} &\equiv P & (\nu x) [x, \sigma :: v] &\equiv \mathbf{0} \\ (\nu l) [l] &\equiv \mathbf{0} & (\nu l) [l]^* &\equiv \mathbf{0} & (\nu l) \langle l \rangle_i &\equiv \mathbf{0} & (\nu l) \langle l \rangle_i^* &\equiv \mathbf{0} & (\nu l) \langle l \rangle_i^\nabla &\equiv \mathbf{0} \\ (\nu u) (\nu u') P &\equiv (\nu u') (\nu u) P & P \mid (\nu u) Q &\equiv (\nu u) (P \mid Q) & (u \notin \text{fn}(P)) \end{aligned} $	

■ **Figure 4** Syntax of the Process language (top) and Structural Congruence for Stores (bottom).

work, we present the **GoldyLocks** language (**GoL** for short), used as a subset of processes of the Go language, and the behavioural types used to model mutual-exclusion locks and shared memory primitives. We then use this calculus and its types to tackle lock liveness and safety, as well as another form of safety: *data race detection*. Our further extension to channels (§ 7) enables us to detect the errors caused by a mixture of shared memory and message passing concurrency. The formulation of a happens-before relation and classification of a data race with respect to the Go memory model along with static analysis of this kind is, to the best of our knowledge, the first of its kind, at least for Go and its mixed memory management features.

Through type soundness and progress theorems of our behavioural typing system (§ 4, § 5), we are able to represent properties of processes by those of types in the modal μ -calculus (§ 6). In this paper, we explore in particular the formal relationship between type-level properties given by the modal μ -calculus and process properties: we prove which subsets of GoL satisfy the properties of the types characterised by the modal μ -calculus (Theorem 30).

We also present a static analysis tool based on the theory. The tool infers from Go programs [3] the memory accesses, locks and message-passing primitives as behavioural types, and generates a μ -calculus model from these types [2]. We then apply the mCRL2 model checker [8] to detect blocking and non-blocking concurrency errors (§ 8). We conclude the paper with an overview of related works (§ 9).

Detailed proofs and additional material can be found in the full version of the paper [13]. The tool and benchmark are available from [2, 1, 3].

2 GoL: a Memory-Aware Core Language for Go

This section introduces a core language that models shared memory concurrency, dubbed **GoldyLocks** (simple subset of Go with shared memory primitives and locks only), shortened as **GoL**. GoL supports two key features for shared memory concurrency: (1) *shared variables*, created by a shared variable creation primitive, whose values can be read from and written to by multiple threads; and (2) *locks* and *read-write locks* (rwlocks) are modelled by creating a lock store, and recording how it is accessed by (read-)lock and (read-)unlock calls.

2.1 Syntax of GoL

The syntax of the calculus, together with the standard structural congruence $P \equiv P'$ (which includes \equiv_α), is given in Figure 4, where e, e' range over *expressions*, x, y over *variables*, l, l' over *locks*, u, u' over *identifiers* (either shared variables or locks) and v over *values* (either local variables, natural numbers or booleans). We write $\tilde{e}, \tilde{v}, \tilde{x}$ and \tilde{u} for a list of expressions, values, variables and names respectively, and use \cdot as the concatenation operator.

Process syntax (P, Q, R, \dots) is given as follows. The *prefix* $\mu; P$ contains either (1) a *silent action* τ ; (2) a *store action* of e in \tilde{x} , $\text{store}(x, e)$; (3) a *load action* of x , bound to y in the continuation, $y \leftarrow \text{load}(x)$; and (4) actions (ℓ) for lock/unlock and read-lock/unlock on program locks (denoted by l).

There are three constructs for “new”: a *new variable* process $\text{new}(x : \sigma); P$ creates a new shared variable in the heap with payload type σ , binding it to x in the continuation P ; a *new lock* process $\text{newl}(l); P$ creates a new program lock and $\text{newrwl}(l); P$ creates a new program read-write lock, binding them to l in the continuation. The syntax includes the *conditional* **if** e **then** P **else** Q , *parallel* process $P \mid Q$, and the *inactive* process $\mathbf{0}$ (often omitted).

A Go program is modelled as a program \mathbf{P} in GoL, written $\{D_i\}_{i \in I}$ in P , which consists of a set of mutually recursive process definitions which encode the goroutines and functions used in the program, together with a process P that encodes the program entry point (**main**). The entry point is usually modelled as $X_0\langle \rangle$, a call to a defined process X_0 . The entry point is the main process in a collection of mutually recursive process definitions (ranged over by D), parametrised by a list of (expressions and locks) variables.

Process variable X is bound by *definition* D of the form of $X(\tilde{x}) = P$ where $\text{fn}(D) = \emptyset$. This is used by *process call* $X\langle \tilde{e}, \tilde{u} \rangle$ which denotes an instance of the process definition bound to X , with formal parameters instantiated to \tilde{e} and \tilde{u} . Note that the entry point could take parameters, if the programmer wants the program to depend on user input data for example, but our examples never make use of that capability.

The part of the syntax denoted by the stores is *runtime* constructs which are generated during the execution (i.e. not written by the programmer and appearing as standalone parallel terms): a shared variable store $[x, \sigma :: v]$ contains message v of type σ ; and we represent five internal states of lock stores, situated on the last line of the left column, where the index i is used for rwlocks and the superscripts \star and \blacktriangledown respectively denote locked and waiting locks. *Restriction* $(\nu u)P$ denotes the runtime handle u for a lock or shared variable bound in P , and thus hidden from external processes.

Finally, the notation $\text{fn}(P)$ denotes the sets of free names (locks, shared variables, local variables), i.e. ones that have not been bound by a restriction operator (νu) , a definition D , a “new” construct, or a load action.

► **Example 1** (Processes from Figure 1 and Figure 2). The following process represents the code in Figure 1. We first separate the **main** function in two parts: the part that instantiates the variable and lock, and spawns the side process in parallel to the continuation, that we call X_0 ; and the rest that processes in parallel to the second goroutine that we put in a separate process P . Process Q is the representation of function **f**, that is run in the second goroutine.

$$\mathbf{P}_{\text{race}} := \left\{ \begin{array}{l} X_0 = \text{new}(x : \text{int}); \text{newrwl}(l); (P\langle x, l \rangle \mid Q\langle x, l \rangle) \\ P(y, z) = \text{rlock}(z); t_1 \leftarrow \text{load}(y); \text{store}(y, t_1 + 10); \text{runlock}(z); \\ \quad \text{rlock}(z); t_2 \leftarrow \text{load}(y); \tau; \text{runlock}(z); \mathbf{0} \\ Q(y, z) = \text{rlock}(z); t_0 \leftarrow \text{load}(y); \text{store}(y, t_0 + 20); \text{runlock}(z); \mathbf{0} \end{array} \right\} \text{ in } X_0\langle \rangle$$

The next process represents the code in Figure 2 in the same fashion as above.

$$P_{\text{safe}} := \left\{ \begin{array}{l} X_0 = \text{new}(x : \text{int}); \text{newrw}(l); (P\langle x, l \rangle \mid Q\langle x, l \rangle) \\ P(y, z) = \text{lock}(z); t_1 \leftarrow \text{load}(y); \text{store}(y, t_1 + 10); \text{unlock}(z); \\ \quad \text{rlock}(z); t_2 \leftarrow \text{load}(y); \tau; \text{runlock}(z); \mathbf{0} \\ Q(y, z) = \text{lock}(z); t_0 \leftarrow \text{load}(y); \text{store}(y, t_0 + 20); \text{unlock}(z); \mathbf{0} \end{array} \right\} \text{ in } X_0 \langle \rangle$$

2.2 Operational Semantics

The semantics of GoL is given by the labelled transition system (LTS) shown in Figure 5. The LTS system enables us to give a simple and uniform definition of barbs in Definition 5 and a formal correspondence with the modal μ -calculus described in § 6. The LTS rules are written $P \xrightarrow{\alpha} P'$, where α is a label of the form:

$$\boxed{\begin{array}{l} \alpha := o_1 \mid o_m, e \quad \iota := * \mid 1.\iota \mid 2.\iota \quad o_m := r\langle x \rangle \mid \overline{r\langle x \rangle} \mid (w\langle x \rangle, \iota) \mid \overline{w\langle x \rangle} \\ o_1 := l\langle l \rangle \mid ul\langle l \rangle \mid rl\langle l \rangle \mid rul\langle l \rangle \mid \ulcorner l \urcorner \mid \ulcorner l \urcorner^* \mid \llcorner l \llcorner \mid \llcorner l \llcorner^\blacktriangledown \mid \llcorner l \llcorner^\blacktriangle \mid \tau_u \mid \tau \quad o := o_m \mid o_1 \end{array}}$$

They can be either a *data-dependent action* o_m along with its data e , used for synchronisation purposes on actions that transmit data, or a *data-independent action* o_1 alone, used for synchronisation on actions that do not transmit meaningful data, and for the synchronisations τ_u and silent action τ .

The actions in o_m define $r\langle x \rangle$ (read), $(w\langle x \rangle, \iota)$ (write), $\overline{r\langle x \rangle}$ and $\overline{w\langle x \rangle}$ (dual actions) of a shared variable x , where ι denotes an *occurrence* (a position in the parallel composition) that is a string of 1s, 2s and $*$. The actions in o_1 define (1) $l\langle l \rangle$ (lock), $ul\langle l \rangle$ (unlock), $rl\langle l \rangle$ (read-lock) and $rul\langle l \rangle$ (read-unlock); (2) lock store actions, $\ulcorner l \urcorner$, $\ulcorner l \urcorner^*$, $\llcorner l \llcorner$, $\llcorner l \llcorner^\blacktriangledown$ and $\llcorner l \llcorner^\blacktriangle$ (whose purpose is to interact with each action in (1) to produce the lock synchronisation τ_l); as well as (3) synchronisations τ_u and silent actions.

► **Remark 2. (1)** The write action $(w\langle x \rangle, \iota)$ uses occurrence ι to denote the position of the thread which contains that action. By using occurrences, we can differentiate two writes on the same variable happening at the same time, and thereby formally define the notion of data race (see Definition 8); and **(2)** one lock store can produce several different actions which then produce lock synchronisation τ_l with different lock primitives. This allows us to implement the properties with mCRL2 straightforwardly, cf. § 8.

We also define the general label o for actions, which only contains action markers and no data, and will be of use for data-independent marking later on, such as barbs. *Occurrences* are ranged over by ι, ι', \dots , where $*$ denotes the empty occurrence, while $1.\iota$ (resp. $2.\iota$) denotes the left (resp. right) shift of ι . The left and right shifting operators on action α , $\text{left}(\alpha)$ and $\text{right}(\alpha)$, are defined as:

$$\text{left}((w\langle x \rangle, \iota), e) = (w\langle x \rangle, 1.\iota), e \quad \text{and} \quad \text{right}((w\langle x \rangle, \iota), e) = (w\langle x \rangle, 2.\iota), e$$

with $\text{left}(\alpha) = \text{right}(\alpha) = \alpha$ if $\alpha \neq (w\langle x \rangle, \iota), e$. Example 3 will explain the use of these operators with the LTS rules.

This LTS defines the semantics of shared variables, locks, and read-write locks which closely follow the specifications in [19]. We first highlight the operational semantics of locks from [17] and rwlocks from [18]. A **lock** is a mutual exclusion lock. It must not be copied after its first use: a lock l is created by $[\text{NEWM}]$, which is guaranteed fresh by the “ (νl) ” operation. It is then locked by $[\text{C-LCK}]$ and unlocked by $[\text{C-ULCK}]$. A **read-write lock** (rwlock) is a reader/writer mutual exclusion lock. The lock can be held by an arbitrary number of readers or a single writer. The zero value for a rwlock is an unlocked state. If a

Lock and Memory actions	Synchronisation rules	
[LCK] $\text{lock}(l); P \xrightarrow{l(l)} P$	$\text{[C-LD]} \frac{P \xrightarrow{r(x), \bar{v}} P' \quad Q \xrightarrow{\overline{r(x)}, v} Q'}{P \mid Q \xrightarrow{\tau_x} P' \mid Q'}$ $\text{[C-ST]} \frac{P \xrightarrow{w(x), \iota, e} P' \quad Q \xrightarrow{\overline{w(x)}, v} Q' \quad e \downarrow v}{P \mid Q \xrightarrow{\tau_x} P' \mid Q'}$ $\text{[C-LCK]} \frac{P \xrightarrow{l(l)} P' \quad Q \xrightarrow{r(l)} Q'}{P \mid Q \xrightarrow{\tau_l} P' \mid Q'}$ $\text{[C-ULCK]} \frac{P \xrightarrow{ul(l)} P' \quad Q \xrightarrow{r(l)^*} Q'}{P \mid Q \xrightarrow{\tau_l} P' \mid Q'}$ $\text{[C-RLCK]} \frac{P \xrightarrow{rl(l)} P' \quad Q \xrightarrow{ul(l)} Q'}{P \mid Q \xrightarrow{\tau_l} P' \mid Q'}$ $\text{[C-RULCK]} \frac{P \xrightarrow{rul(l)} P' \quad Q \xrightarrow{ul(l)^*} Q'}{P \mid Q \xrightarrow{\tau_l} P' \mid Q'}$ $\text{[C-WAIT]} \frac{P \xrightarrow{l(l)} P' \quad Q \xrightarrow{ul(l)^*} Q'}{P \mid Q \xrightarrow{\tau} P \mid Q'}$ $\text{[TAU]} \tau; P \xrightarrow{\tau} P$	
[ULCK] $\text{unlock}(l); P \xrightarrow{ul(l)} P$		
[RLCK] $\text{rlock}(l); P \xrightarrow{rl(l)} P$		
[RULCK] $\text{runlock}(l); P \xrightarrow{rul(l)} P$		
[LOAD] $y \leftarrow \text{load}(x); P \xrightarrow{r(x), v} P \{v/y\}$		
[STO] $\text{store}(x, e); P \xrightarrow{w(x), *, e} P$		
[M-LCK] $[l] \xrightarrow{r(l)} [l]^*$		
[M-ULCK] $[l]^* \xrightarrow{r(l)^*} [l]$		
[RW-LCK] $\langle l \rangle_0^* \xrightarrow{r(l)} \langle l \rangle_0^*$		
[RW-ULCK] $\langle l \rangle_0^* \xrightarrow{r(l)^*} \langle l \rangle_0$		
[RW-RLCK] $\langle l \rangle_i \xrightarrow{ul(l)} \langle l \rangle_{i+1}$		
[RW-RULCK] $\langle l \rangle_{i+1} \xrightarrow{ul(l)^*} \langle l \rangle_i$		
[RW-WAIT] $\langle l \rangle_i \xrightarrow{ul(l)^*} \langle l \rangle_i$		
[RW-WULCK] $\langle l \rangle_{i+1}^* \xrightarrow{ul(l)^*} \langle l \rangle_i^*$		
[H-LD] $[x, \sigma :: v] \xrightarrow{r(x), v} [x, \sigma :: v]$		
[H-ST] $[x, \sigma :: v] \xrightarrow{w(x), v'} [x, \sigma :: v']$		
[NEWV] $\text{new}(y : \sigma); P \xrightarrow{\tau} (\nu y) (P \mid [y, \sigma :: \perp])$		
[NEWRWM] $\text{newrl}(l); P \xrightarrow{\tau} (\nu l) (P \mid [l])$	[NEWRWM] $\text{newrw}(l); P \xrightarrow{\tau} (\nu l) (P \mid \langle l \rangle_0)$	
[PAR-L] $\frac{P \xrightarrow{\alpha} P'}{P \mid Q \xrightarrow{\text{left}(\alpha)} P' \mid Q}$	[PAR-R] $\frac{Q \xrightarrow{\alpha} Q'}{P \mid Q \xrightarrow{\text{right}(\alpha)} P \mid Q'}$	[RES1] $\frac{P \xrightarrow{\alpha} P' \quad u \notin \text{fn}(\alpha)}{(\nu u)P \xrightarrow{\alpha} (\nu u)P'}$
[IFT] $\frac{e \downarrow \text{true}}{\text{if } e \text{ then } P \text{ else } Q \xrightarrow{\tau} P}$	[IFF] $\frac{e \downarrow \text{false}}{\text{if } e \text{ then } P \text{ else } Q \xrightarrow{\tau} Q}$	[RES2] $\frac{P \xrightarrow{\tau_u} P'}{(\nu u)P \xrightarrow{\tau} (\nu u)P'}$
[DEF] $\frac{P \{ \bar{v}, \bar{u} / \bar{x} \} \mid Q \xrightarrow{\alpha} R \quad e_i \downarrow v_i \quad X(\bar{x}) = P \in \{D_i\}_{i \in I}}{X(\bar{e}, \bar{u}) \mid Q \xrightarrow{\alpha} R}$	[ALPHA] $\frac{P \equiv_{\alpha} P' \quad P' \xrightarrow{\beta} P''}{P \xrightarrow{\beta} P''}$	

■ **Figure 5** LTS Reduction Semantics for the Processes.

goroutine holds a rwlock for reading and another goroutine calls **Lock**, no goroutine should expect to be able to acquire a read-lock until both the initial read-lock and the staged **Lock** call are released. This is to ensure that the lock eventually becomes available to writers; a blocked **Lock** call excludes new readers from acquiring the lock. To model this situation, we annotate a freshly created rwlock by the counter i (instanciated at 0 by [NEWRWM]); this counter is incremented by any fired read-lock (by [C-RLCK]), and blocked from increasing if a **Lock** action gets staged (by [C-WAIT], **note how the Lock action is not consumed by this rule**); then it is unlocked by read-unlock calls (by [C-RULCK]) until the pending number of read-locks becomes 0, and finally write-locked (by [C-LCK]) and further unlocked by the corresponding unlock (by [C-ULCK]), if a **Lock** was previously staged by [C-WAIT].

A *shared variable* is implemented at runtime by a named area in the store, which stores a value of its payload data type, and that can be written to or read by any process within its scope. It is created by [NEWV] with an initial value for declared type σ (0 for int, false for bool, etc.), accessed for reading by [C-LD] and for writing by [C-ST].

The $[\text{PAR-}*]$ rules are explained in Example 3 below.

► **Example 3 (Occurrences).** Let $P = \text{store}(x, e); P'$, $Q = \text{store}(x, e'); Q'$ and $R = z \leftarrow \text{load}(x); R'$. It follows $P \xrightarrow{(\mathbf{w}\langle x \rangle, *) , e} P'$, $Q \xrightarrow{(\mathbf{w}\langle x \rangle, *) , e'} Q'$ and $R \xrightarrow{r\langle x \rangle, v} R' \{v/z\}$.

If we compose P and Q , we use $[\text{PAR-L}]$ and $[\text{PAR-R}]$ to determine the new reductions:

$$\begin{array}{ll} P \mid Q \xrightarrow{(\mathbf{w}\langle x \rangle, 1.*), e} P' \mid Q & \text{left } ((\mathbf{w}\langle x \rangle, *) , e) = (\mathbf{w}\langle x \rangle, 1.*), e \\ P \mid Q \xrightarrow{(\mathbf{w}\langle x \rangle, 2.*), e'} P \mid Q' & \text{right } ((\mathbf{w}\langle x \rangle, *) , e') = (\mathbf{w}\langle x \rangle, 2.*), e' \end{array}$$

Composing again, with R :

$$\begin{array}{ll} (P \mid Q) \mid R \xrightarrow{(\mathbf{w}\langle x \rangle, 1.1.*), e} (P' \mid Q) \mid R & \text{left } ((\mathbf{w}\langle x \rangle, 1.*), e) = (\mathbf{w}\langle x \rangle, 1.1.*), e \\ (P \mid Q) \mid R \xrightarrow{(\mathbf{w}\langle x \rangle, 1.2.*), e'} (P \mid Q') \mid R & \text{left } ((\mathbf{w}\langle x \rangle, 2.*), e') = (\mathbf{w}\langle x \rangle, 1.2.*), e' \\ (P \mid Q) \mid R \xrightarrow{r\langle x \rangle, v} (P \mid Q) \mid R' \{v/z\} & \text{right } (r\langle x \rangle, v) = r\langle x \rangle, v \end{array}$$

For process definitions, we implicitly assume the existence of an ambient set of definitions $\{D_i\}_{i \in I}$. Rule $[\text{DEF}]$ replaces X by the corresponding process definition (according to the underlying definition environment), instantiating the parameters accordingly. The remaining rules are standard from process calculus literature [36]. We define \rightarrow as $\equiv \xrightarrow{\tau} \equiv \cup \equiv \xrightarrow{\tau_u} \equiv$.

We define a *normal form* for terms, which is used later in § 6:

► **Definition 4 (Normal Form).** A term P is *in normal form* if $P = (\nu \tilde{u})P'$ and $P' \not\equiv (\nu u)P''$.

We note that, with structural congruence, every well-formed term can be transformed to normal form, and we can then study reduction up to normal form, in order to witness synchronisation actions on channels, memory and mutex.

3 Defining Safety and Liveness: Data Race and Happens-Before

We define the properties of data race freedom and lock safety/liveness through *barbs* (§ 3.1). A **data race** happens when two writers (or a reader and a writer) can concurrently access the same shared variable at the same time. **Unsafe lock access** happens if (1) unlock happens before lock happens or before waiting read-unlocks release the lock; or (2) read-unlock happens before read-lock happens or after a lock call accesses the process lock. **Lock liveness** identifies the ability of (read-)lock requests to always eventually fire. Our first main result is a formalisation of the *happens-before* relation and other properties specified in the Go memory model [16] and a correspondence between a data race characterisation through the happens before relation and another characterisation of a data race through barbs.

3.1 Safety and Liveness Properties through Barbs

We first define barbed process predicates [32] introducing predicates for locks and shared variable accesses. The predicate $P \downarrow_o$ means that P immediately offers a visible action o .

► **Definition 5 (Process barbs).** The barbs are defined as follows:

Prefix Actions: $\text{store}(x, e) \downarrow_{(\mathbf{w}\langle x \rangle, *)}$; $y \leftarrow \text{load}(x) \downarrow_{r\langle x \rangle}$; $\text{lock}(l) \downarrow_{l(l)}$;
 $\text{unlock}(l) \downarrow_{ul(l)}$; $\text{rlock}(l) \downarrow_{rl(l)}$; $\text{runlock}(l) \downarrow_{rul(l)}$

Programs: if $P \xrightarrow{o, e} P'$ where $o = o_m$ is an action over a shared variable, or $P \xrightarrow{o} P'$ where $o = o_l$ is τ_u or a lock action, then $P \downarrow_o$.

$$\boxed{
\begin{array}{c}
\text{(CON)} \frac{\mu \downarrow_o \quad P \downarrow_{o'}}{\mu; P \triangleright o \mapsto o'} \quad \text{(TRA)} \frac{P \triangleright o \mapsto o' \quad P \triangleright o' \mapsto o''}{P \triangleright o \mapsto o''} \quad \text{(RED)} \frac{P \rightarrow^* P' \quad P' \triangleright o \mapsto o'}{P \triangleright o \mapsto o'} \\
\text{(PAR-L)} \frac{P \triangleright o \mapsto o'}{P \mid Q \triangleright \text{left}(o) \mapsto \text{left}(o')} \quad \text{(PAR-R)} \frac{Q \triangleright o \mapsto o'}{P \mid Q \triangleright \text{right}(o) \mapsto \text{right}(o')} \\
\text{(U-L)} \frac{P \downarrow_{l\langle l \rangle} \quad P \downarrow_{ul\langle l \rangle}}{P \triangleright ul\langle l \rangle \mapsto l\langle l \rangle} \quad \text{(RU-L)} \frac{P \downarrow_{l\langle l \rangle} \quad P \downarrow_{rul\langle l \rangle}}{P \triangleright rul\langle l \rangle \mapsto l\langle l \rangle} \\
\text{(U-RL)} \frac{P \downarrow_{rl\langle l \rangle} \quad P \downarrow_{ul\langle l \rangle}}{P \triangleright ul\langle l \rangle \mapsto rl\langle l \rangle} \quad \text{(L-RL)} \frac{P \downarrow_{rl\langle l \rangle} \quad P \downarrow_{l\langle l \rangle}}{P \triangleright l\langle l \rangle \mapsto rl\langle l \rangle} \\
\text{(RES)} \frac{P \triangleright o \mapsto o' \quad u \notin \text{fn}(o) \cup \text{fn}(o')}{(\nu u)P \triangleright o \mapsto o'} \quad \text{(ALPHA)} \frac{P \triangleright o \mapsto o' \quad P \equiv_\alpha Q}{Q \triangleright o \mapsto o'}
\end{array}
}$$

We omit the symmetric rules for most rules ending in a parallel process $P \mid Q$.

■ **Figure 6** Happens-Before Relation.

Actions in this case are the same ones as defined before in the operational semantics of GoL, expect for silent action τ . We write $P \downarrow_o$ if $P \rightarrow^* P'$ and $P' \downarrow_o$.

We first define a safety property for locks in Definition 6.

► **Definition 6 (Safety).** Program P is *safe* if for all P such that $P \rightarrow^* (\nu \tilde{u})P$, (a) if $P \downarrow_{ul\langle l \rangle}$ then $P \downarrow_{rl\langle l \rangle}$; and (b) if $P \downarrow_{rul\langle l \rangle}$ then $P \downarrow_{l\langle l \rangle}$.

Safety states that in all reachable program states, the unlock action will happen only if the process lock is already locked by the lock action; and the read-unlock will happen only if the process lock is locked by the read-lock action.

Next we define the liveness property: all (read-)lock requests will always eventually fire (i.e. perform a synchronisation).

► **Definition 7 (Liveness).** Program P is *live* if for all P such that $P \rightarrow^* (\nu \tilde{u})P$, if $P \downarrow_{l\langle l \rangle}$ or $P \downarrow_{rl\langle l \rangle}$ then $P \downarrow_{\tau}$.

3.2 Happens Before and Data Race

We now define the happens-before relation, closely following [16], and investigate its relationship with data races. The *happens-before* relation between actions o and o' , denoted by $P \triangleright o \mapsto o'$, is defined in Figure 6. It is a binary relation which is transitive, non-reflexive and non-symmetric, where $o, o' \in \{\mathbf{w}\langle x \rangle, \iota, \mathbf{r}\langle x \rangle, l\langle l \rangle, ul\langle l \rangle, rl\langle l \rangle, rul\langle l \rangle\}$. The operation $\text{left}(o)$ denotes that occurrence ι in o changes to $1.\iota$, defined as before by $\text{left}(\mathbf{w}\langle x \rangle, \iota) = (\mathbf{w}\langle x \rangle, 1.\iota)$; otherwise $\text{left}(o) = o$. The rules follow the specification in [16].

Rule (CON) specifies that within a single goroutine, the happens-before order is the order expressed by the program. Rule (RED) gives a form of inheritance: if P reduces to P' and P' has an order between two actions, then P accepts this order as valid as well, as it is a possible future. However, if $P \triangleright o \mapsto o'$, it does not necessarily hold for all of P 's reductions.

Rule (PAR-L) replaces $(\mathbf{w}\langle x \rangle, \iota)$ with $(\mathbf{w}\langle x \rangle, 1.\iota)$ if o or o' is a write action. Rule (PAR-R) is symmetric. Rules (U-L), (RU-L), (U-RL) and (L-RL) specify the ordering between (read)locks and (read)unlocks, following the reduction semantics.

The following definition states that if a write action happens concurrently with another write action or a read action to the same variable, the program has a data-race.

► **Definition 8** (Data Race). Program P has a *data race* if there exist two distinct actions $o_1 \neq o_2$, two distinct occurrences $\iota \neq \iota'$, and $P \rightarrow^* (\nu \tilde{u})P$, with $o_1 = (w\langle x \rangle, \iota)$ and $o_2 \in \{(w\langle x \rangle, \iota'), r\langle x \rangle\}$, such that $P \Downarrow_{o_1}$, $P \Downarrow_{o_2}$, $\neg(P \triangleright o_1 \mapsto o_2)$ and $\neg(P \triangleright o_2 \mapsto o_1)$. Program P is *data race free* if it has no data race.

The following theorem states that the data race defined with the happens-before relation coincides with the characterisation given by barbs. The proof is by induction, see [13].

► **Theorem 9** (Characterisation of Data Race). P has a data race if and only if there exists P such that $P \rightarrow^* (\nu \tilde{u})P$ with $P \Downarrow_{o_1}$, $P \Downarrow_{o_2}$, $o_1 = (w\langle x \rangle, \iota)$, $o_2 \in \{(w\langle x \rangle, \iota'), r\langle x \rangle\}$ and $\iota \neq \iota'$.

► **Example 10** (Processes from Figure 1). We show a possible reduction of P_{race} in Example 1 that causes the (bad) race.

$$\begin{aligned} P_{\text{race}} &= \text{new}(x : \text{int}); \text{newrw}(l); \left(\begin{array}{l} \text{rlock}(l); t_1 \leftarrow \text{load}(x); \text{store}(x, t_1 + 10); \text{runlock}(l); \\ \text{rlock}(l); t_2 \leftarrow \text{load}(x); \tau; \text{runlock}(l); \mathbf{0} \\ | \text{rlock}(l); t_0 \leftarrow \text{load}(x); \text{store}(x, t_0 + 20); \text{runlock}(l); \mathbf{0} \end{array} \right) \\ &\rightarrow^2 (\nu xl) \left(\begin{array}{l} \text{rlock}(l); t_1 \leftarrow \text{load}(x); \text{store}(x, t_1 + 10); \text{runlock}(l); \\ \text{rlock}(l); t_2 \leftarrow \text{load}(x); \tau; \text{runlock}(l); \mathbf{0} \\ | \text{rlock}(l); t_0 \leftarrow \text{load}(x); \text{store}(x, t_0 + 20); \text{runlock}(l); \mathbf{0} \mid [x, \text{int} :: 0] \mid \langle l \rangle_0 \end{array} \right) \\ &\rightarrow^6 (\nu xl) \left(\begin{array}{l} \text{store}(x, 10); \text{runlock}(l); \text{rlock}(l); t_2 \leftarrow \text{load}(x); \tau; \text{runlock}(l); \mathbf{0} \\ | \text{store}(x, 20); \text{runlock}(l); \mathbf{0} \mid [x, \text{int} :: 0] \mid \langle l \rangle_2 \end{array} \right) = (\nu xl)P' \end{aligned}$$

Note that the first line is obtained by rewriting using the process definition structure and the $[\text{DEF}]$ rule, that tells us the rewritten program and the program with calls share the same reductions. Then we have $P' \Downarrow_{(w\langle x \rangle, 1.1.1.*)}$ and $P' \Downarrow_{(w\langle x \rangle, 1.1.2.*)}$, hence P_{race} has a data race.

On the other hand, P_{safe} is data race free, which is ensured by checking every reduction chain of the process for the absence of data race.

4 A Behavioural Typing System for GoL

Our typing system introduces types for locks and shared memory, representing the status of runtime processes accessing to shared variables. It serves as a behavioural abstraction of a valid GoL program, where types take the form of CCS processes with name creation.

4.1 Behavioural Types with Shared Variables and Mutexes

The syntax of types (T, S, \dots) and the structural congruence for the types are given in Figure 7. The type $\vartheta; T$ denotes a store $w(u)$, load $r(u)$ of shared variable u , lock $l(l)$, unlock $ul(l)$, rlock $rl(l)$, runlock $rul(l)$ of a (rw)lock l , followed by the behaviour denoted by type T . It also includes an explicit silent action τ followed by the behaviour T_P .

The type constructs x^\blacksquare , $[l]$, $[l]^\star$, $\langle l \rangle_i$, $\langle l \rangle_i^\star$ and $\langle l \rangle_i^\blacktriangledown$ denote the type representations of runtime shared variable, unlocked and locked locks, unlocked (or read-locked), locked and lock-waiting rwlocks, respectively. Types for variables and locks include shared variable and (rw)lock creation $\text{new}(x); T$, $\text{newl}(l); T$ and $\text{newrw}(l); T$ which respectively bind x and l in T . $\text{fn}(T)$ denotes the set of free names of type T .

4.2 Typing System with Shared Variables and Mutexes

Our typing system is defined in Figure 8.

The judgement $(\Gamma \vdash P \blacktriangleright T)$, where Γ is a typing environment that maintains information about locks and shared variables, and types the part of a term explicitly written by the developer. We write $\Gamma \vdash \mathcal{J}$ for $\mathcal{J} \in \Gamma$ and $\Gamma \vdash e : \sigma$ to state that the expression e is well-typed

$ \begin{array}{l} T, S := \vartheta; T \mid (T \mid S) \mid \mathbf{0} \mid (\nu u)T \\ \mid \oplus\{T_i\}_{i \in I} \mid \mathbf{t}_X \langle \tilde{u} \rangle \mid \mathbf{new}(x); T \\ \mid \mathbf{newl}(l); T \mid \mathbf{newrw}(l); T \\ \mid x^\blacksquare \mid [l] \mid [l]^* \\ \mid \langle l \rangle_i \mid \langle l \rangle_i^* \mid \langle l \rangle_i^\nabla \end{array} $	$ \begin{array}{l} \mathbf{T} := \{\mathbf{t}_i(\tilde{y}_i) = T_i\}_{i \in I} \text{ in } T \\ \vartheta := \tau \mid \mathbf{r}(x) \\ \mid \mathbf{w}(x) \mid \xi \\ \xi := \mathbf{l}(l) \mid \mathbf{ul}(l) \\ \mid \mathbf{rl}(l) \mid \mathbf{rul}(l) \end{array} $
<hr/> $ \begin{array}{l} T \mid S \equiv S \mid T \quad T \mid (S \mid S') \equiv (T \mid S) \mid S' \quad T \mid \mathbf{0} \equiv T \quad (\nu x)x^\blacksquare \equiv \mathbf{0} \\ (\nu l)[l] \equiv \mathbf{0} \quad (\nu l)[l]^* \equiv \mathbf{0} \quad (\nu l)\langle l \rangle_i \equiv \mathbf{0} \quad (\nu l)\langle l \rangle_i^* \equiv \mathbf{0} \quad (\nu l)\langle l \rangle_i^\nabla \equiv \mathbf{0} \\ (\nu u)(\nu u')T \equiv (\nu u')(\nu u)T \quad T \mid (\nu u)S \equiv (\nu u)(T \mid S) \quad (u \notin \text{fn}(T)) \end{array} $	

■ **Figure 7** Syntax of the types.

according to the types of variables in Γ . We write $u:t$ for the typing of a name in generality, which can be (1) $x:\text{var}(\sigma)$ to denote a shared variable x with stored value type σ and (2) $l:\text{Lock}$ to state that l is a (rw)lock. We omit the rules of expressions e . We write $\text{dom}(\Gamma)$ to denote the set of locks and shared variable bindings in Γ .

The rules are as follows. Rules $\langle \text{LOAD} \rangle$ and $\langle \text{STO} \rangle$ type load and store types for shared variable x where the type of the stored value matches the payload type σ of value x , and the continuation P has type T . Rules $\langle \text{LCK} \rangle$ and $\langle \text{ULCK} \rangle$ (and $\langle \text{RLCK} \rangle$ and $\langle \text{RULCK} \rangle$) type the lock actions in processes by corresponding types. There is no payload type to check, only that the lock name is associated to a lock or read-write lock. Rules $\langle \text{NEWV} \rangle$ and $\langle \text{NEWM} \rangle$ (resp. $\langle \text{NEWRW} \rangle$) allocate a fresh shared variable name with payload type σ or a lock (resp. rwlock). Other context rules are standard.

The judgement $(\Gamma \vdash_B P \blacktriangleright T)$ types process created during execution of a program and provides the invariants to prove the type safety. B is a set of shared variables and locks with associated runtime buffers to ensure their uniqueness. A shared variable heap is typed with rule $\langle \text{HEAP} \rangle$, and all five states of locks are typed by corresponding lock types. Restriction is typed here, as it takes the relevant type out of the typing context and removes the corresponding name from B .

The judgement $(\Gamma \vdash_B \mathbf{P} \blacktriangleright \mathbf{T})$ types a program, that consists of a process and a set of runtime stores, accordingly to their respective types.

We use the structural congruence on types to define *normal forms* of types in the same way as done for GoL terms in Definition 4, and study further properties on types up to normal form. Examples of typing of processes can be found in Example 11.

► **Example 11.** The unsafe program of Figure 1, modelled by process \mathbf{P}_{race} in Example 1, has the following type:

$$\mathbf{T}_{\text{race}} := \left\{ \begin{array}{l}
\mathbf{t}_0 = \mathbf{new}(x); \mathbf{newrw}(l); (\mathbf{t}_P \langle x, l \rangle \mid \mathbf{t}_Q \langle x, l \rangle) \\
\mathbf{t}_P(y, z) = \mathbf{rl}(z); \mathbf{r}(y); \mathbf{w}(y); \mathbf{rul}(z); \mathbf{rl}(z); \mathbf{r}(y); \tau; \mathbf{rul}(z); \mathbf{0} \\
\mathbf{t}_Q(y, z) = \mathbf{rl}(z); \mathbf{r}(y); \mathbf{w}(y); \mathbf{rul}(z); \mathbf{0}
\end{array} \right\} \text{ in } \mathbf{t}_0 \langle \rangle$$

The safe version in Figure 2, modelled by process \mathbf{P}_{safe} in Example 1, has type:

$$\mathbf{T}_{\text{safe}} := \left\{ \begin{array}{l}
\mathbf{t}_0 = \mathbf{new}(x); \mathbf{newrw}(l); (\mathbf{t}_P \langle x, l \rangle \mid \mathbf{t}_Q \langle x, l \rangle) \\
\mathbf{t}_P(y, z) = \mathbf{l}(z); \mathbf{r}(y); \mathbf{w}(y); \mathbf{ul}(z); \mathbf{rl}(z); \mathbf{r}(y); \tau; \mathbf{rul}(z); \mathbf{0} \\
\mathbf{t}_Q(y, z) = \mathbf{l}(z); \mathbf{r}(y); \mathbf{w}(y); \mathbf{ul}(z); \mathbf{0}
\end{array} \right\} \text{ in } \mathbf{t}_0 \langle \rangle$$

4.3 Operational Semantics of the Behavioural Types

This section defines the semantics of our types. The labels, ranged over by o, o' , have the form:

$\boxed{\Gamma \vdash P \blacktriangleright T}$	$\langle \text{ZERO} \rangle \frac{}{\Gamma \vdash \mathbf{0} \blacktriangleright \mathbf{0}}$	$\langle \text{NEWV} \rangle \frac{\Gamma, x : \text{var}(\sigma) \vdash P \blacktriangleright T}{\Gamma \vdash \text{new}(x : \sigma); P \blacktriangleright \text{new}(x); T}$
$\langle \text{NEWM} \rangle \frac{\Gamma, l : \text{Lock} \vdash P \blacktriangleright T}{\Gamma \vdash \text{newl}(l); P \blacktriangleright \text{newl}(l); T}$	$\langle \text{NEWRW} \rangle \frac{\Gamma, l : \text{Lock} \vdash P \blacktriangleright T}{\Gamma \vdash \text{newrw}(l); P \blacktriangleright \text{newrw}(l); T}$	
$\langle \text{LCK} \rangle \frac{\Gamma \vdash l : \text{Lock} \quad \Gamma \vdash P \blacktriangleright T}{\Gamma \vdash \text{lock}(l); P \blacktriangleright l(l); T}$	$\langle \text{STO} \rangle \frac{\Gamma \vdash x : \text{var}(\sigma) \quad \Gamma \vdash e : \sigma \quad \Gamma \vdash P \blacktriangleright T}{\Gamma \vdash \text{store}(x, e); P \blacktriangleright w(x); T}$	
$\langle \text{ULCK} \rangle \frac{\Gamma \vdash l : \text{Lock} \quad \Gamma \vdash P \blacktriangleright T}{\Gamma \vdash \text{unlock}(l); P \blacktriangleright ul(l); T}$	$\langle \text{LOAD} \rangle \frac{\Gamma \vdash x : \text{var}(\sigma) \quad \Gamma, y : \text{var}(\sigma) \vdash P \blacktriangleright T}{\Gamma \vdash y \leftarrow \text{load}(x); P \blacktriangleright r(x); T}$	
$\langle \text{RLCK} \rangle \frac{\Gamma \vdash l : \text{Lock} \quad \Gamma \vdash P \blacktriangleright T}{\Gamma \vdash \text{rlock}(l); P \blacktriangleright rl(l); T}$	$\langle \text{RULCK} \rangle \frac{\Gamma \vdash l : \text{Lock} \quad \Gamma \vdash P \blacktriangleright T}{\Gamma \vdash \text{runlock}(l); P \blacktriangleright rul(l); T}$	
$\langle \text{TAU} \rangle \frac{\Gamma \vdash P \blacktriangleright T}{\Gamma \vdash \tau; P \blacktriangleright \tau; T}$	$\langle \text{VAR} \rangle \frac{\Gamma \vdash \tilde{e} : \tilde{\sigma} \quad \Gamma \vdash \tilde{u} : \tilde{t}}{\Gamma, X(\tilde{\sigma}, \tilde{t}) \vdash X(\tilde{e}, \tilde{u}) \blacktriangleright \mathbf{t}_X(\tilde{u})}$	
$\langle \text{PAR} \rangle \frac{\Gamma \vdash P \blacktriangleright T \quad \Gamma \vdash Q \blacktriangleright S}{\Gamma \vdash P \mid Q \blacktriangleright (T \mid S)}$	$\langle \text{SEL} \rangle \frac{\Gamma \vdash e : \text{bool} \quad \Gamma \vdash P \blacktriangleright T \quad \Gamma \vdash Q \blacktriangleright S}{\Gamma \vdash \text{if } e \text{ then } P \text{ else } Q \blacktriangleright \oplus\{T, S\}}$	
$\boxed{\Gamma \vdash_B P \blacktriangleright T}$	$\langle \text{MUT} \rangle \frac{\Gamma \vdash l : \text{Lock}}{\Gamma \vdash_{\{l\}} [l] \blacktriangleright [l]}$	$\langle \text{L-M} \rangle \frac{\Gamma \vdash l : \text{Lock}}{\Gamma \vdash_{\{l\}} [l]^* \blacktriangleright [l]^*}$
$\langle \text{L-RW} \rangle \frac{\Gamma \vdash l : \text{Lock}}{\Gamma \vdash_{\{l\}} \langle l \rangle_i^* \blacktriangleright \langle l \rangle_i^*}$	$\langle \text{W-RW} \rangle \frac{\Gamma \vdash l : \text{Lock}}{\Gamma \vdash_{\{l\}} \langle l \rangle_i^\nabla \blacktriangleright \langle l \rangle_i^\nabla}$	$\langle \text{HEAP} \rangle \frac{\Gamma \vdash x : \text{var}(\sigma)}{\Gamma \vdash_{\{x\}} [x, \sigma :: v] \blacktriangleright x^\blacksquare}$
$\langle \text{RES} \rangle \frac{\Gamma, u : t \vdash_B P \blacktriangleright T}{\Gamma \vdash_{B \setminus u} (\nu u) P \blacktriangleright (\nu u) T}$	$\langle \text{PARR} \rangle \frac{\Gamma \vdash_{B_1} P \blacktriangleright T \quad \Gamma \vdash_{B_2} Q \blacktriangleright S \quad B_1 \cap B_2 = \emptyset}{\Gamma \vdash_{B_1 \cup B_2} P \mid Q \blacktriangleright (T \mid S)}$	
$\boxed{\Gamma \vdash_B P \blacktriangleright T}$	$\langle \text{DEF} \rangle \frac{\Gamma, X_i(\tilde{\sigma}_i, \tilde{t}_i), \tilde{x}_i : \tilde{\sigma}_i, \tilde{y}_i : \tilde{t}_i \vdash P_i \blacktriangleright T_i \quad \Gamma, X_1(\tilde{\sigma}_1, \tilde{t}_1), \dots, X_n(\tilde{\sigma}_n, \tilde{t}_n) \vdash Q \blacktriangleright S}{\Gamma \vdash \{X_i(\tilde{x}_i, \tilde{y}_i) = P_i\}_{i \in I} \text{ in } Q \blacktriangleright \{\mathbf{t}_{X_i}(\tilde{y}_i) = T_i\}_{i \in I} \text{ in } S}$	

■ **Figure 8** Typing Rules for Shared Variables and Mutexes.

$$\boxed{o := r\langle x \rangle \mid (w\langle x, \iota \rangle \mid l\langle l \rangle \mid ul\langle l \rangle \mid rl\langle l \rangle \mid rul\langle l \rangle \mid x^\blacksquare \mid \ulcorner l \urcorner \mid \ulcorner l \urcorner^* \mid \llcorner l \llcorner \mid \llcorner l \llcorner^\nabla \mid \llcorner l \llcorner^\blacktriangle \mid \tau \mid \tau_u}$$

The labels denote the actions introduced in this paper: load and store actions, lock, unlock, rlock and runlock actions, shared heap manipulation, and the five kinds of (rw)lock state transitions. The end of the line is for silent transition and synchronisation over a name.

The semantics of our types is given by the labelled transition system (LTS) (modulo α -conversion), extending that of CCS, which is shown in Figure 9.

Rules $[\text{STO}]$ and $[\text{LOAD}]$ allow a type to emit a store and load action on a shared variable x . Rule $[\text{LCK}]$ (resp. $[\text{ULCK}]$) emits a lock (resp. unlock) action on a shared lock l . Rules $[\text{NEWM}]$ and $[\text{NEWRW}]$ (resp. $[\text{NEWV}]$) create a new shared heap x or unlocked lock (resp. rwlock) store l . Rule $[\text{HEAP}]$ models the ability of a shared heap to be read or updated at any time, and rule $[\text{C-HEAP}]$ allows a load or store action to synchronise with its associated heap.

Rule $[\text{M-LCK}]$ makes a lock to be closed, and rule $[\text{M-ULCK}]$ unlocks a claimed lock. Rules $[\text{C-LCK}]$ and $[\text{C-ULCK}]$ make the corresponding actions to synchronise with their associated lock store. Equivalent rules for rwlocks act the same as in the processes. Pay attention to the same quirk as in processes: $[\text{C-WAIT}]$ does not consume the lock action in T , as this rule serves to forbid further read-lock calls from being executed if a lock call is staged.

<div style="border: 1px solid black; padding: 2px; margin-bottom: 5px;">Lock and Memory actions</div> <div style="padding: 2px;"> $\text{ LCK } \quad l(l); T \xrightarrow{l(l)} T$ $\text{ ULCK } \quad \text{ul}(l); T \xrightarrow{\text{ul}(l)} T$ $\text{ RLCK } \quad \text{rl}(l); T \xrightarrow{\text{rl}(l)} T$ $\text{ RULCK } \quad \text{rul}(l); T \xrightarrow{\text{rul}(l)} T$ $\text{ LOAD } \quad r(x); T \xrightarrow{r(x)} T$ $\text{ STO } \quad \text{w}(x); T \xrightarrow{\text{w}(x,*)} T$ </div> <hr/> <div style="padding: 2px;"> $\text{ M-LCK } \quad [l] \xrightarrow{r(l)} [l]^*$ $\text{ M-ULCK } \quad [l]^* \xrightarrow{r(l)^*} [l]$ $\text{ RW-LCK } \quad \langle l \rangle_0 \xrightarrow{r(l)} \langle l \rangle_0^*$ $\text{ RW-ULCK } \quad \langle l \rangle_0^* \xrightarrow{r(l)^*} \langle l \rangle_0$ $\text{ RW-RLCK } \quad \langle l \rangle_i \xrightarrow{\text{rl}(l)} \langle l \rangle_{i+1}$ $\text{ RW-RULCK } \quad \langle l \rangle_{i+1} \xrightarrow{\text{rul}(l)} \langle l \rangle_i$ $\text{ RW-WAIT } \quad \langle l \rangle_i \xrightarrow{\text{ul}(l)} \langle l \rangle_i^*$ $\text{ RW-WULCK } \quad \langle l \rangle_{i+1}^* \xrightarrow{\text{ul}(l)^*} \langle l \rangle_i$ $\text{ HEAP } \quad x^\blacksquare \xrightarrow{x} x^\blacksquare$ </div>	<div style="border: 1px solid black; padding: 2px; margin-bottom: 5px;">Synchronisation rules</div> <div style="padding: 2px;"> $\text{ C-HEAP } \quad \frac{T \xrightarrow{o} T' \quad S \xrightarrow{x^\blacksquare} S' \quad o = (\text{w}(x), l), r(x)}{T \mid S \xrightarrow{\tau_x} T' \mid S'}$ $\text{ C-LCK } \quad \frac{T \xrightarrow{l(l)} T' \quad S \xrightarrow{r(l)} S'}{T \mid S \xrightarrow{\tau_l} T' \mid S'}$ $\text{ C-ULCK } \quad \frac{T \xrightarrow{\text{ul}(l)} T' \quad S \xrightarrow{r(l)^*} S'}{T \mid S \xrightarrow{\tau_l} T' \mid S'}$ $\text{ C-RLCK } \quad \frac{T \xrightarrow{\text{rl}(l)} T' \quad S \xrightarrow{\text{ul}(l)} S'}{T \mid S \xrightarrow{\tau_l} T' \mid S'}$ $\text{ C-RULCK } \quad \frac{T \xrightarrow{\text{rul}(l)} T' \quad S \xrightarrow{\text{ul}(l)^*} S'}{T \mid S \xrightarrow{\tau_l} T' \mid S'}$ $\text{ C-WAIT } \quad \frac{T \xrightarrow{l(l)} T' \quad S \xrightarrow{\text{ul}(l)} S'}{T \mid S \xrightarrow{\tau_l} T' \mid S'}$ $\text{ TAU } \quad \tau; T \xrightarrow{\tau} T$ </div>
<div style="border: 1px solid black; padding: 2px; margin-bottom: 5px;">Runtime structures creation</div> <div style="padding: 2px;"> $\text{ NEWV } \quad \text{new}(x); T \xrightarrow{\tau} (\nu x) (T \mid x^\blacksquare)$ $\text{ NEWM } \quad \text{newl}(l); T \xrightarrow{\tau} (\nu l) (T \mid [l])$ $\text{ NEWRWM } \quad \text{newrwl}(l); T \xrightarrow{\tau} (\nu l) (T \mid \langle l \rangle_0)$ </div>	
<div style="border: 1px solid black; padding: 2px; margin-bottom: 5px;">Context rules</div> <div style="padding: 2px;"> $\text{ ALPHA } \quad \frac{T \equiv T' \quad T' \xrightarrow{o} T''}{T \xrightarrow{o} T''}$ $\text{ RES1 } \quad \frac{T \xrightarrow{o} T' \quad u \notin \text{fn}(o)}{(\nu u)T \xrightarrow{o} (\nu u)T'}$ $\text{ RES2 } \quad \frac{T \xrightarrow{\tau_u} T'}{(\nu u)T \xrightarrow{\tau} (\nu u)T'}$ $\text{ PAR-L } \quad \frac{j \in I \quad \bigoplus \{T_i\}_{i \in I} \xrightarrow{\tau} T_j}{T \xrightarrow{o} T'}$ $\text{ PAR-R } \quad \frac{S \xrightarrow{o} S'}{T \mid S \xrightarrow{\text{right}(o)} T \mid S'}$ $\text{ DEF } \quad \frac{T \{ \tilde{u}/\tilde{x} \} \mid S \xrightarrow{o} T' \quad \mathbf{t}_X(\tilde{x}) = T}{\mathbf{t}_X(\tilde{u}) \mid S \xrightarrow{o} T'}$ </div>	

■ **Figure 9** LTS Reduction Semantics for the Types.

Rule |SEL| represents the internal choice behaviour of the conditional processes.

In Figure 9, we omit the symmetric rules for parallel composed processes (such as |C-HEAP|). We write \rightarrow for $\equiv \xrightarrow{\tau} \equiv \cup \equiv \xrightarrow{\tau_u} \equiv$ and $T \rightarrow^* T' \xrightarrow{o} T''$ if there exist T' and T'' such that $T \rightarrow^* T' \xrightarrow{o} T''$.

► **Example 12.** The unsafe version of Figure 1, modelled by process \mathbf{P}_{race} in Example 1 and typed by \mathbf{T}_{race} in Example 11, has the following possible reduction (following the same reduction order as Example 10):

$$\begin{aligned}
\mathbf{T}_{\text{race}} &= \text{new}(x); \text{newrwl}(l); \left(\begin{array}{l} \text{rl}(l); r(x); \text{w}(x); \text{rul}(l); \text{rl}(l); r(x); \tau; \text{rul}(l); \mathbf{0} \\ \text{rl}(l); r(x); \text{w}(x); \text{rul}(l); \mathbf{0} \end{array} \right) \\
&\rightarrow^2 (\nu xl) \left(\begin{array}{l} \text{rl}(l); r(x); \text{w}(x); \text{rul}(l); \text{rl}(l); r(x); \tau; \text{rul}(l); \mathbf{0} \\ \text{rl}(l); r(x); \text{w}(x); \text{rul}(l); \mathbf{0} \mid x^\blacksquare \mid \langle l \rangle_0 \end{array} \right) \\
&\rightarrow^6 (\nu xl) \left(\begin{array}{l} \text{w}(x); \text{rul}(l); \text{rl}(l); r(x); \tau; \text{rul}(l); \mathbf{0} \\ \text{w}(x); \text{rul}(l); \mathbf{0} \mid x^\blacksquare \mid \langle l \rangle_2 \end{array} \right) = (\nu xl)T'
\end{aligned}$$

We note that T' is a type of P' which has a data race in Example 10.

5 Properties of GoL Processes and Types

This section proves two main results, the subject reduction and progress properties with respect to behavioural types. Our goal is to classify subsets of GoL programs for which liveness, data race freedom and safety coincide with liveness, data race freedom and safety of their types. Detailed proofs for this section are available in [13].

5.1 Type soundness of GoL processes

A basic property for types is to be preserved under structural congruence and to be able to reduce the same as the process.

► **Proposition 13** (Subject Congruence). If $\Gamma \vdash_B P \blacktriangleright T$ and $P \equiv P'$, then $\exists T' \equiv T$ such that $\Gamma \vdash_B P' \blacktriangleright T'$.

The following type soundness theorem shows that behaviours of processes can be simulated by behaviours of types.

► **Theorem 14** (Subject Reduction). If $\Gamma \vdash_B P \blacktriangleright T$ and $P \rightarrow P'$, then $\exists T'$ such that $\Gamma \vdash_B P' \blacktriangleright T'$ and $T \rightarrow T'$.

The following progress theorem says that the action availability on types infers that on processes.

We first need to define *barbs* to represent capabilities of a type at a given time in reduction, akin to how process barbs are defined in Definition 5.

► **Definition 15** (Type Barbs). The barbs on types are defined as follows:

Prefix Actions: $w(x) \downarrow_{(w\langle x, * \rangle)}$; $r(x) \downarrow_{r\langle x \rangle}$; $l(l) \downarrow_{l\langle l \rangle}$;
 $ul(l) \downarrow_{ul\langle l \rangle}$; $rl(l) \downarrow_{rl\langle l \rangle}$; $rul(l) \downarrow_{rul\langle l \rangle}$

Types: if $T \xrightarrow{o} T'$ where o is a communication action over a shared variable or τ_u or a lock action, then $T \downarrow_o$.

► **Theorem 16** (Progress). Suppose $\Gamma \vdash P \blacktriangleright T$. Then if $T \xrightarrow{o} T_0$ for $o \in \{\tau_u, \tau\}$ for some heap or lock u , then there exists P', T' such that $P \rightarrow P', T \xrightarrow{o} T'$, and $\Gamma \vdash P' \blacktriangleright T'$.

To prove this theorem, we use a lemma which shows a correspondence of barbs between processes and types (defined similarly with barbs of processes, cf Definition 15). Note that in Theorem 16, T' and T_0 might be different. This is because a selection type (i.e. the internal choice) can reduce non-deterministically but the corresponding conditional process usually is deterministic.

5.2 Safety and Liveness for Types

In this subsection, we define safety and liveness for types, which correspond to Definitions 6, 7 and 8, respectively.

► **Definition 17** (Safety). Type \mathbf{T} is *safe* if for all T such that $\mathbf{T} \rightarrow^* (\nu \tilde{u})T$, (a) if $T \downarrow_{ul\langle l \rangle}$ then $T \downarrow_{r\langle l \rangle}$; and (b) if $T \downarrow_{rul\langle l \rangle}$ then $T \downarrow_{l\langle l \rangle}$.

► **Definition 18** (Liveness). Type \mathbf{T} is *live* if for all T such that $\mathbf{T} \rightarrow^* (\nu \tilde{u})T$, if $T \downarrow_{l\langle l \rangle}$ or $T \downarrow_{rl\langle l \rangle}$ then $T \downarrow_{\tau_l}$.

► **Definition 19** (Data Race). \mathbf{T} has a data race if and only if there exists T such that $\mathbf{T} \rightarrow^* (\nu \tilde{u})T$ with $T \downarrow_{o_1}, T \downarrow_{o_2}, o_1 = (w\langle x, \iota \rangle), o_2 \in \{(w\langle x, \iota' \rangle), r\langle x \rangle\}$ and $\iota \neq \iota'$.

We say that \mathbf{T} is data race free if it has no data race.

5.3 Liveness and Safety for Typed GoL

In this section, we state several propositions and theorems adapted from [27] to our new process and types primitives and their LTSs. Our goal is to classify subsets of GoL programs for which liveness, data race freedom and safety coincide with liveness, data race freedom and safety of their types.

First, we prove that safety and data race freedom (which is a form of safety) have no restriction, and that proving that a type is safe always entails the associated program is safe.

► **Theorem 20** (Process Safety and Data Race Freedom). *Suppose $\Gamma \vdash \mathbf{P} \blacktriangleright \mathbf{T}$ and \mathbf{T} is safe (resp. data race free). Then \mathbf{P} is safe (resp. data race free).*

We then prove that liveness of types is equivalent to liveness of programs for a subset of the GoL programs, in three steps: (1) programs that always have a terminating path, (2) finite branching programs, and (3) programs that simulate non-deterministic branching in infinitely recurring conditionals.

We first study the case of programs that always have a path to termination:

► **Definition 21** (May Converging Program). Let $\Gamma \vdash \mathbf{P} \blacktriangleright \mathbf{T}$. We write $\mathbf{P} \in \text{May}\downarrow$ if for all $\mathbf{P} \rightarrow^* \mathbf{P}'$, $\mathbf{P}' \rightarrow^* \mathbf{0}$.

An example of May Converging program is the following program, where process P loops and alternates x to values 1 and 0 until the *end* flag is set, and Q loops reading x until it reads a value 0, in which case it sets the *end* flag and returns:

$$\mathbf{P}_{\text{mc}} := \left\{ \begin{array}{l} X_0 = \text{new}(x : \text{int}); \text{new}(\text{end} : \text{bool}); \text{newrw}(l); \\ \quad (P\langle x, \text{end}, l \rangle \mid Q\langle x, \text{end}, l \rangle) \\ P(x, \text{end}, l) = \text{lock}(l); y \leftarrow \text{load}(x); \text{store}(x, 1 - y); z \leftarrow \text{load}(\text{end}); \\ \quad \text{unlock}(l); \text{if } z \text{ then } \mathbf{0} \text{ else } P\langle x, \text{end}, l \rangle \\ Q(x, \text{end}, l) = \text{lock}(l); y \leftarrow \text{load}(x); \text{unlock}(l); \\ \quad \text{if } y = 0 \text{ then } \text{lock}(l); \text{store}(\text{end}, \text{true}); \text{unlock}(l); \mathbf{0} \\ \quad \text{else } Q\langle x, \text{end}, l \rangle \end{array} \right\} \text{ in } X_0 \langle \rangle$$

The next proposition states that on these programs, proving liveness of their types is enough to ensure liveness of the associated program.

► **Proposition 22.** *Assume $\Gamma \vdash \mathbf{P} \blacktriangleright \mathbf{T}$ and \mathbf{T} is live. (1) Suppose there exists \mathbf{P}' such that $\mathbf{P} \rightarrow^* \mathbf{P}' \not\rightarrow$. Then $\mathbf{P}' \equiv \mathbf{0}$; and (2) If $\mathbf{P} \in \text{May}\downarrow$, then \mathbf{P} is live.*

We now need to define a subset of May Converging programs, that is the set of always terminating programs. This is needed because our implementation, that we describe in § 8, only allows to check and ensure liveness for terminating programs, ie. the result of our tool for liveness is assured to coincide with actual program liveness only on terminating programs.

Note that the tool is able to model check non-terminating programs (under the assumption they don't spawn an unbounded amount of new threads), but may in rare instances lead to a false positive, due to the approximations the model checker has to make in this case.

► **Definition 23** (Terminating Program). We write $\mathbf{P} \in \text{Terminate}$ if there exists some non-negative number n such that, for all \mathbf{P} such that $\mathbf{P} \rightarrow^n \mathbf{P}$, $\mathbf{P} \not\rightarrow$.

The following proposition states that this subset of programs is included in the set of May Converging programs. We note that this inclusion is strict: a program that may loop forever on a select construct, with a timeout branch that terminates the program, is May Converging but not terminating in the sense of the above definition, as we may always find a reduction path that continues longer than any finite bound.

► **Proposition 24.** $P \in \text{Terminate}$ implies $P \in \text{May}\Downarrow$.

Proof. By definition of the May Converging set of programs, all programs that always converge are May Converging. ◀

► **Example 25.** Note that the running examples we defined in Figure 1 and 2 are both terminating, and so are their modelling processes given in Example 1.

The next set of programs we highlight is finite branching programs. We first define a series of items, including deterministic marking of conditionals and the set of infinitely branching programs, in order to grab everything not infinitely branching (*ie.* outside of the defined set).

Marked Programs. Given a program P we define its *marking*, written $\text{mark}(P)$, as the program obtained by deterministically labelling every occurrence of a conditional of the form **if** e **then** P **else** Q in P , as **if** ^{n} e **then** P **else** Q , such that n is distinct natural number for all conditionals in P .

Marked Reduction Semantics. We modify the marked reduction semantics, written $P \xrightarrow{l} P'$, stating that program P reduces to P' in a single step, performing action l . The grammar of action labels is defined as: $l := \alpha \mid n \cdot L \mid n \cdot R$ where α denotes a non-conditional action, taking into account all existing actions and all rules expect $[\text{IFT}]$ and $[\text{IFF}]$, $n \cdot L$ denotes a conditional branch marked with the natural number n in which the **then** branch is chosen, and $n \cdot R$ denotes a conditional branch in which the **else** branch is chosen. Because of the changes in notations, conditional branches are not considered a standard reduction step in \rightarrow any more. The marked reduction semantics replace rules $[\text{IFT}]$ and $[\text{IFF}]$.

Trace. We define an execution trace of a program P as the potentially infinite sequence of action labels \vec{l} such that $P \xrightarrow{l_1} P_1 \xrightarrow{l_2} \dots$, with $\vec{l} = \{l_1, l_2, \dots\}$. We write \mathbb{T}_P for the set of all possible traces of a process P .

Reduction Contexts are given by: $\mathbb{C}_r := [] \mid (P \mid \mathbb{C}_r) \mid (\mathbb{C}_r \mid P) \mid (\nu u)\mathbb{C}_r$.

Infinite Conditional. We say that P has infinite conditionals, written as $P \in \text{Inf}$, iff $\text{mark}(P) \rightarrow^* \mathbb{C}_r[\text{if}^n e \text{ then } P \text{ else } Q] = R$, for some n , and R has an infinite trace where $n \cdot L$ or $n \cdot R$ appears infinitely often. We say that such an n is an *infinite conditional mark* and write $\text{InfCond}(P)$ for the set of all such marks.

We state in the next proposition that finite branching programs can be ensured live by checking for liveness of their types.

► **Proposition 26** (Liveness for Finite Branching). *Suppose $\Gamma \vdash P \blacktriangleright T$ and T is live and $P \notin \text{Inf}$. Then P is live.*

An example of finite branching program is the Dining Philosophers problem:

$$P_{\text{dinephil}} := \left\{ \begin{array}{l} X_0 \\ = \text{new}(f_1 : \text{int}); \text{new}(f_2 : \text{int}); \text{new}(f_3 : \text{int}); \\ \text{newl}(l_1); \text{newl}(l_2); \text{newl}(l_3); \\ \left(\begin{array}{l} P\langle f_1, f_2, l_1, l_2, 1 \rangle \mid P\langle f_2, f_3, l_2, l_3, 2 \rangle \\ \mid P\langle f_1, f_3, l_1, l_3, 3 \rangle \end{array} \right) \\ P(f_l, f_r, l_l, l_r, id) = \text{lock}(l_l); y \leftarrow \text{load}(f_l); \tau; \text{store}(f_l, id); \\ \text{lock}(l_r); z \leftarrow \text{load}(f_r); \tau; \text{store}(f_r, id + 2); \\ \text{unlock}(l_r); \text{unlock}(l_l); P\langle f_l, f_r, l_l, l_r, id \rangle \end{array} \right\} \text{ in } X_0 \langle \rangle$$

Here, P defines the behaviour of a philosopher, trying to get a hold of both forks assigned to him, and then release them. Other implementations of this problem's algorithm (including ones using channel communications) can be found in the full version [13].

Next we define in the infinite branching programs a subset containing only programs that simulate non-deterministic branching.

Conditional Mapping. The mapping $(\mathbf{P})^*$ replaces all occurrences of marked conditionals $\mathbf{if}^n e \mathbf{then} P \mathbf{else} Q$, such that $n \in \text{InfCond}(\mathbf{P})$, with $\mathbf{if} * \mathbf{then} P \mathbf{else} Q$. Its reduction semantics follow the nondeterministic semantics of selection in types, reducing with a τ label. This mapping is applicable to processes P .

Alternating Conditionals. We say that \mathbf{P} has *alternating conditional branches*, written $\mathbf{P} \in \text{AC}$, iff $\mathbf{P} \in \text{Inf}$ and if $\mathbf{P} \rightarrow^* (\nu \tilde{u})P$ then $\mathbf{P}^* \Downarrow_o$ implies $P \Downarrow_o$.

The concurrent version of the Prime Sieve [27, 33] is an example of program that has alternating conditionals. Our implementation of it in Go can be found in the full version [13], and is not detailed here as it uses channels, which we will introduce in an extension to this work in § 7. An other simple example of alternating conditionals is as follows:

$$\mathbf{P}_{\text{ac}} := \left\{ \begin{array}{l} X_0 = \text{new}(x : \text{bool}); \text{new}(y : \text{int}); P\langle x, y \rangle \\ P\langle b, i \rangle = z \leftarrow \text{load}(b); \mathbf{if} z \mathbf{then} t \leftarrow \text{load}(i); \\ \quad \text{store}(i, t + 1); \text{store}(b, \text{not}(z)); P\langle b, i \rangle \mathbf{else} \text{store}(b, \text{not}(z)); P\langle b, i \rangle \end{array} \right\} \text{ in } X_0\langle \rangle$$

We finally state that programs in the alternating conditionals set can be ensured live by ensuring that their types are live.

► **Theorem 27 (Liveness).** *Suppose $\Gamma \vdash \mathbf{P} \blacktriangleright \mathbf{T}$ and \mathbf{T} is live and $\mathbf{P} \in \text{AC}$. Then \mathbf{P} is live.*

To summarise this section, we identified three classes of GoL programs for which we can prove liveness by proving type liveness: (1) programs that always have access to a terminating path (Definition 21 and Proposition 22), including the strict subset of programs that always terminate within a finite number of reduction steps, similar to our running examples; (2) programs that do not exhibit an infinite branch containing an infinitely occurring conditional (Proposition 26), such as the Dining Philosophers problem (used in our benchmarks, see § 8 for more details); and (3) programs with infinite branches that contain infinitely occurring conditionals, with the condition that these infinitely occurring conditionals simulate a non-deterministic choice (Theorem 27), like our Prime Sieve implementation [27, 33] and the example presented above.

6 Verifying Program Properties: the Modal μ -Calculus

In this section, we introduce the modal μ -calculus and express various properties over the types. We then explain how the type-level properties are transposed to process-level properties, as proved in § 5.3.

6.1 The Modal μ -Calculus

We first define a *pointed LTS* for the types, to denote the capabilities available at this point in the simulation.

► **Definition 28 (Pointed LTS of types).** We define the pointed LTS of a program's types as:

A set of states \mathcal{S} , labelled by the (restriction-less) types accessible by reducing from the entrypoint \mathbf{t}_0 with $\xrightarrow{\tau}$ and $\xrightarrow{\tau_u}$; this entrypoint is defined as the type of the entrypoint X_0 of the program: $\mathcal{S} := \{T : \mathbf{t}_0 \rightarrow^* (\nu \tilde{u})T \text{ and } T \not\equiv (\nu \tilde{u}')T'\}$.

A set of labelled transitions \mathcal{A} , in $\mathcal{S} \times \mathcal{S} \times \{\tau, \tau_u\}$: $\mathcal{A} := \{(T, T', o) : T, T' \in \mathcal{S} \text{ and } T \xrightarrow{o} T'\}$.

A set of **barbs** attached to each state, describing the actions its labelled type can take according to the set of barbs of this type. These take the form of the barbs as they were defined above: $\forall T \in \mathcal{S}, \mathcal{F}(T) := \{\downarrow_o : T \downarrow_o\}$.

The modal μ -calculus is a calculus that allows to express temporal properties on such pointed LTS, like the fact that there exists an accessible state where some property is true, or the fact that some property is true in *all reachable states*. The syntax of these formulae is given below, where α is a set of barbs over the types available to the LTS of types, or transition actions τ or τ_u available as transitions to the LTS of types, as defined above:

$$\begin{array}{l} \phi := \top \mid \perp \mid \neg\phi \mid \phi \wedge \phi \mid \phi \vee \phi \mid \phi \Rightarrow \phi \mid [\alpha]\phi \mid \langle \alpha \rangle \phi \mid \nu Z.\phi \mid \mu Z.\phi \mid Z \\ \alpha := \alpha + \alpha \mid \downarrow_o \mid \downarrow_{\bar{o}} \mid \tau \mid \tau_u \mid \mathbb{S} \quad \mathbb{S} := \{\tau_u : u \in \text{fn}(T)\} \cup \{\tau\} \end{array}$$

The formulae contain the true and false constants, negation, implication, conjunction and disjunction (both of which can be generalised over a set of actions, where this set can be restrained by some condition).

The *diamond modality*, $\langle \alpha \rangle \phi$, is true when at least one of the actions in α is available from the current state and, if it is a barb then ϕ must be true in the current state, and if it is a transition action then ϕ must be true in the resulting state. If no action in α is available, then this formula is false. For example, $\langle \downarrow_{(w(z),*)} \rangle \top$ holds on every state where a store action on z is available as the main action, but not when the only store action available is labelled otherwise, e.g. $1.*$.

The *box modality*, $[\alpha]\phi$, is valid when, for every state reachable by following an action in α from the current state, ϕ is true. This set of states can be empty, in case no action in α is available, in which case this formula is vacuously true. For example, $[\tau]\perp$ is true only when no τ transition is available to the current state of the pointed LTS of the type.

The *lowest fixed point* $\mu Z.\phi$ and *greatest fixed point* $\nu Z.\phi$ are the standard recursive constructs, where the least fixed point is the intersection of prefixed points, and the greatest fixed point is the union of postfixed points. That implies the following properties, given for understanding:

1. $\mu Z.Z = \perp$: the lowest fixed point defaults to false;
2. $\nu Z.Z = \top$: the greatest fixed point defaults to true;
3. if $\phi[Z := \psi] \Rightarrow \psi$ then $\mu Z.\phi \Rightarrow \psi$: the lowest fixed point can be expanded on the left of a logical implication;
4. if $\psi \Rightarrow \phi[Z := \psi]$ then $\psi \Rightarrow \nu Z.\phi$: the greatest fixed point can be expanded on the right of a logical implication.

To express that some modal μ -calculus formula ϕ is true on a state labelled with type T in the LTS \mathcal{T} , we say that T satisfies ϕ in the LTS \mathcal{T} , written $T \models_{\mathcal{T}} \phi$.

Two key properties that can be expressed are: ϕ is *always true*, which means that every state T in \mathcal{T} satisfies that formula; and ϕ is *eventually true* which means that there exists a reachable state that satisfies this formula. These are expressed with the fixed-point modalities explained above:

$$\text{Always } \phi: \Psi(\phi) = \nu Z.\phi \wedge [-]Z \quad \text{Eventually } \phi: \Phi(\phi) = \mu Z.\phi \vee \langle - \rangle Z$$

6.2 Properties of the Behavioural Types

Figure 10 defines the *local* properties we check on the states of the behavioural types LTS, which means they are defined for *one state only*. The global properties can be checked on the entypoint of the LTS by checking for $\Psi(\phi)$, ie. “always ϕ ”.

1. Mutex safety (a): $\psi_{s_a} = \bigwedge_l \langle \downarrow_{ul}(l) \rangle \top \Rightarrow \langle \downarrow_{rl} \rangle \top$
2. Mutex safety (b): $\psi_{s_b} = \bigwedge_l \langle \downarrow_{rul}(l) \rangle \top \Rightarrow \langle \downarrow_{rl} \rangle \top$
3. Mutex liveness: $\psi_l = \bigwedge_l \langle \downarrow_l + \downarrow_{rl}(l) \rangle \top \Rightarrow \Phi(\langle \tau_l \rangle \top)$
4. Data race freedom: $\psi_d = \bigwedge_{x,\ell} \langle \downarrow_{(w(x),\ell)} \rangle \top \Rightarrow \left[\sum_{\ell' \neq \ell} \downarrow_{(w(x),\ell')} + \downarrow_{r(x)} \right] \perp$

■ **Figure 10** Modal μ -calculus properties of types.

Property ψ_{s_a} checks for the first half of lock safety, that is a lock can only be unlocked if it is currently in locked state, and property ψ_{s_b} checks the second half of lock safety, that is a read/write-lock can only be read-unlocked one level if it is in a read-locked state currently.

Property ψ_l states lock liveness, that is if a lock or read-lock action is staged, the same lock will eventually synchronise (and as such, when applied on a global level $\Psi(\psi_l)$, the lock or read-lock in question will eventually fire, since it becomes false if at any point there is a lock or read-lock staged but no future synchronisation on the lock). Remember that in our model, liveness of the types only entails liveness of the program if the program is in one of the subsets defined previously, in particular if the program terminates or only has alternating conditionals.

Finally, property ψ_d checks local data race freedom, that is if a write action is available on some variable x , then no other read or write action is available on the same variable in the current state. $\Psi(\psi_d)$ checks for data race freedom on the whole of accessible states, so checking that on the entrypoint \mathbf{t}_0 of a type LTS \mathcal{T} ensures the type of the associated program is data race free, and thus that said program is data race free.

► **Example 29.** We can check that the type T' from Example 12 does not verify ψ_d :

$$\psi_d = \left(\langle \downarrow_{(w(z),1.1.1.*)} \rangle \top \Rightarrow [\downarrow_{(w(z),1.1.2.*)} + \downarrow_{r(z)}] \perp \right) \wedge \left(\langle \downarrow_{(w(z),1.1.2.*)} \rangle \top \Rightarrow [\downarrow_{(w(z),1.1.1.*)} + \downarrow_{r(z)}] \perp \right)$$

which is false for T' , hence $T' \not\models_{\mathcal{T}_{\text{race}}} \psi_d$: locally, T' has a datarace. Then $\mathbf{t}_0 \not\models_{\mathcal{T}_{\text{race}}} \Psi(\psi_d)$, meaning \mathbf{T}_{race} has a data race, since its associated entrypoint in its LTS $\mathcal{T}_{\text{race}}$ does not satisfy data race freedom property $\Psi(\psi_d)$.

On the other hand, the type \mathbf{T}_{safe} from Example 12, modelling the safe version of our running example, verifies the data race freedom property, as well as safety and liveness:

$$\mathbf{T}_{\text{safe}} \models_{\mathcal{T}_{\text{safe}}} \Psi(\psi_d) \wedge \Psi(\psi_l) \wedge \Psi(\psi_{s_a} \wedge \psi_{s_b})$$

The types corresponding to the other examples in § 5.3 (\mathbf{P}_{mc} , $\mathbf{P}_{\text{dinephil}}$ and \mathbf{P}_{ac}) are also safe, live and data race free.

The following theorem states that type-level model-checking can justify process properties under the conditions given in § 5.3. We define the pointed LTS of processes $\mathcal{T}_{\mathbf{P}}$ and the satisfaction property $\mathbf{P} \models_{\mathcal{T}_{\mathbf{P}}} \phi$ in the same way as they are defined for types in this section.

► **Theorem 30** (Model Checking of GoL processes). *Suppose $\Gamma \vdash \mathbf{P} \blacktriangleright \mathbf{T}$.*

1. *If $\mathbf{T} \models_{\mathcal{T}_{\mathbf{T}}} \Psi(\phi)$ for $\phi \in \{\psi_{s_a}, \psi_{s_b}, \psi_d\}$, then $\mathbf{P} \models_{\mathcal{T}_{\mathbf{P}}} \Psi(\phi)$.*
2. *If $\mathbf{T} \models_{\mathcal{T}_{\mathbf{T}}} \Psi(\psi_l)$ and either (a) $\mathbf{P} \in \text{May}_{\downarrow}$ or (b) $\mathbf{P} \notin \text{Inf}$ or (c) $\mathbf{P} \in \text{AC}$, then $\mathbf{P} \models_{\mathcal{T}_{\mathbf{P}}} \Psi(\psi_l)$.*

Proof. By Theorems 20 and 27, and Propositions 22 and 26. ◀

```

1 func main() {
2     var x int
3     ch := make(chan int, 1)           ⇒ 2
4     go f(ch, &x)
5     ch <- Lock // send to ch
6     x += 10    // protected by ch ⇒ race
7     <-ch      // receive from ch
8     ch <- Lock
9     fmt.Println("x is", x)
10    <-ch
11 }
12
13 func f(ch chan int, ptr *int) {
14     ch <- Lock
15     *ptr += 20 // protected by ch ⇒ race
16     <-ch
17 }

```

■ **Figure 11** Go programs: safe (size 1) ⇒ race (size 2).

7 Extending the framework for Go with channels

One of the core features of the Go language is the use of channels for communication in concurrent programming. In Go programs, a number of concurrency bugs can be caused by a *mixture* of data races and communication problems. In this section, we develop a theory which can uniformly analyse concurrency errors caused by a mix of shared memory accesses and asynchronous message-passing communications, integrating coherently our framework in [27, 28]. We include channel communications as a synchronisation primitive in our model for data race checking, following the official Go specification.

Figure 11 illustrates a Go program, which makes use of a channel `ch` to synchronise the `main` and `f` functions updating the content of the shared variable `x`. On line 3, the statement `ch := make(chan int, num)` creates a new shared channel `ch` with a buffer size of `num` for passing `int` values. Channels can be sent to or received from using the `<-` operator, where `ch <- value` and `<-ch` depict sending `value` to the channel and receiving from the channel respectively. At runtime, sending to a full channel (i.e. number of items in channel $\geq num$), or receiving from an empty channel (i.e. number of items in channel = 0) blocks. The `go` keyword in front of a function call on line 4 spawns a lightweight thread (called a *goroutine*) to execute the body of function `f`. The two parameters of function `f` – a channel `ch`, and an `int` pointer `ptr` – are shared between the caller and callee goroutines, `main` and `f`. Since concurrent access to the shared pointer `ptr` may introduce a data race, a pair of channel send and receive are used to ensure serialised, mutually exclusive access to `ptr` in `f` and `x` in `main`. If the buffer size of the shared channel is set to 2 by mistake (as denoted by ⇒ in line 3), allowing simultaneous write requests to `x` on lines 6 and 15, the program could output “x is 20” with a bad scheduling, dropping the increase of 10 in the same thread as the print statement. We use this program as our running example in this section.

7.1 Channels in Processes

We add to the processes the following constructs to account for *channel actions* (defined as $\pi := c!(e) \mid c?(x) \mid \tau$) and runtime buffer:

$$P := \dots \mid \pi; P \mid \text{close } c; P \mid \text{select}\{\pi_i; P_i\}_{i \in I} \mid \text{newchan}(c; \sigma, n); P \mid c\langle \sigma, n \rangle :: \tilde{v} \mid c^*\langle \sigma, n \rangle :: \tilde{v}$$

Channels are ranged over by a, b, c , which are from now also included under the generic names u , and sets of channels are ranged over by \tilde{c} . The new syntax contains the ability to send and receive messages through channels, in capabilities under prefix π , and the ability

Channel actions	Synchronisation rules
<div style="border: 1px solid black; padding: 2px; margin-bottom: 5px;">[SND] $c!(e); P \xrightarrow{\bar{c},e} P$</div> <div style="border: 1px solid black; padding: 2px; margin-bottom: 5px;">[RVC] $c?(y); Q \xrightarrow{c,v} Q \{v/y\}$</div> <div style="border: 1px solid black; padding: 2px; margin-bottom: 5px;">[END] $\text{close } c; P \xrightarrow{\text{end}[c]} P$</div> <hr style="border: 0.5px solid black;"/> <div style="border: 1px solid black; padding: 2px; margin-bottom: 5px;">[BUF] $c\langle\sigma, n\rangle::\tilde{v} \xrightarrow{\text{end}[c]} c^*\langle\sigma, n\rangle::\tilde{v}$ $\tilde{v} < n$</div> <div style="border: 1px solid black; padding: 2px; margin-bottom: 5px;">[PUSH] $\frac{c\langle\sigma, n\rangle::\tilde{v} \xrightarrow{\bullet c,v} c\langle\sigma, n\rangle::v \cdot \tilde{v}}{c\langle\sigma, n\rangle::\tilde{v} \cdot v \xrightarrow{\bullet c,v} c\langle\sigma, n\rangle::\tilde{v}}$</div> <div style="border: 1px solid black; padding: 2px; margin-bottom: 5px;">[POP] $c\langle\sigma, n\rangle::\tilde{v} \cdot v \xrightarrow{\bullet c,v} c\langle\sigma, n\rangle::\tilde{v}$</div> <div style="border: 1px solid black; padding: 2px; margin-bottom: 5px;">[CPOP] $c^*\langle\sigma, n\rangle::\tilde{v} \cdot v \xrightarrow{c^*,v} c^*\langle\sigma, n\rangle::\tilde{v}$</div> <div style="border: 1px solid black; padding: 2px; margin-bottom: 5px;">[CLD] $c^*\langle\sigma, n\rangle::\emptyset \xrightarrow{c^*,\perp\sigma} c^*\langle\sigma, n\rangle::\emptyset$</div>	<div style="border: 1px solid black; padding: 2px; margin-bottom: 5px;">[CLOSE] $\frac{P \xrightarrow{\text{end}[c]} P' \quad Q \xrightarrow{\overline{\text{end}[c]}} Q'}{P \mid Q \xrightarrow{\tau} P' \mid Q'}$</div> <div style="border: 1px solid black; padding: 2px; margin-bottom: 5px;">[SCOM] $\frac{P \xrightarrow{\bar{c},e} P' \quad Q \xrightarrow{c,v} Q' \quad e \downarrow v}{(P \mid Q) \mid c\langle\sigma, 0\rangle::\emptyset \xrightarrow{\tau_c} (P' \mid Q') \mid c\langle\sigma, 0\rangle::\emptyset}$</div> <div style="border: 1px solid black; padding: 2px; margin-bottom: 5px;">[OUT] $\frac{P \xrightarrow{\bar{c},e} P' \quad Q \xrightarrow{\bullet c,v} Q' \quad e \downarrow v}{P \mid Q \xrightarrow{\tau_c} P' \mid Q'}$</div> <div style="border: 1px solid black; padding: 2px; margin-bottom: 5px;">[IN] $\frac{P \xrightarrow{c,v} P' \quad Q \xrightarrow{\bullet c,v} Q' \text{ or } Q \xrightarrow{c^*,v} Q'}{P \mid Q \xrightarrow{\tau_c} P' \mid Q'}$</div> <div style="border: 1px solid black; padding: 2px; margin-bottom: 5px;">[BRA] $\frac{\pi_j; P_j \mid P \xrightarrow{\alpha} P' \quad \alpha \in \{\tau, \tau_c\}}{\text{select}\{\pi_i; P_i\}_{i \in I} \mid P \xrightarrow{\alpha} P'}$</div>
<div style="border: 1px solid black; padding: 2px; display: inline-block;">Runtime creation</div>	<div style="border: 1px solid black; padding: 2px; display: inline-block;">[NEWC] $\text{newchan}(y:\sigma, n); P \xrightarrow{\tau} (\nu c) (P \{c/y\} \mid c\langle\sigma, n\rangle::\emptyset) \quad (c \notin \text{fn}(P))$</div>

■ **Figure 12** Remaining LTS Semantics of Processes.

to close a channel. There is also a **select** construct that allows selection between several processes guarded by channel send or receive actions, or a silent action. Lastly, we can create a new channel, and there are two runtime constructs denoting respectively open and closed channel c with payload type σ , allowed buffer size n and current buffered messages \tilde{v} .

We add the structural congruence rules for queues, $(\nu c)c\langle\sigma, n\rangle::\tilde{v} \equiv \mathbf{0}$ and $(\nu c)c^*\langle\sigma, n\rangle::\tilde{v} \equiv \mathbf{0}$, and to the LTS the new corresponding reduction rules, along with their labels, shown in Figure 12. The rules include creating a new channel with [NEWC]; sending to and receiving from a buffered channel with [OUT] and [IN]; closing a channel with [CLOSE]; synchronous communications for channels with buffer size 0 using rule [SCOM]; and reducing a select construct with [BRA].

► **Example 31** (Processes from Figure 11). The following process represents the unsafe version of the code in Figure 11. As in Example 1, we separate the **main** function in two parts, the part that instantiates the variable and channel, and spawns the side process in parallel to the continuation; and two called processes P and Q .

$$P_{c\text{-race}} := \left\{ \begin{array}{l} X_0 = \text{new}(x : \text{int}); \text{newchan}(c:\text{int}, 2); (P\langle x, c \rangle \mid Q\langle x, c \rangle) \\ P(y, z) = z!(\text{Lock}); t_1 \leftarrow \text{load}(y); \text{store}(y, t_1 + 10); z?(u_1); \\ \quad z!(\text{Lock}); t_2 \leftarrow \text{load}(y); \tau; z?(u_2); \mathbf{0} \\ Q(y, z) = z!(\text{Lock}); t_0 \leftarrow \text{load}(y); \text{store}(y, t_0 + 20); z?(u_0); \mathbf{0} \end{array} \right\} \text{ in } X_0 \langle \rangle$$

The safe version $P_{c\text{-safe}}$ is the same, replacing the 2 for a 1 in the channel instantiation. This example reduces, like the one with a rwlock, allowing to see the possible data race:

$$P_{c\text{-race}} \rightarrow^6 (\nu xl) \left(\begin{array}{l} \text{store}(x, 10); c?(u_1); \\ c!(\text{Lock}); t_2 \leftarrow \text{load}(x); \tau; c?(u_2); \mathbf{0} \\ \text{store}(x, 20); c?(u_0); \mathbf{0} \mid [x, \text{int} :: 0] \mid c\langle \text{int}, 2 \rangle::\text{Lock} \cdot \text{Lock} \end{array} \right) = (\nu xl) P'$$

7.2 Liveness and Safety for Channels

To define the liveness and safety properties for channels, we first extend the barbs as follows:

► **Definition 32** (Process barbs). The barbs are expanded as follows:

prefix actions: $c?(x) \downarrow_c$; $c!(e) \downarrow_{\bar{c}}$.

select: we add the rule:
$$\frac{\forall i \in \{1, \dots, n\} : \pi_i; P_i \xrightarrow{o_i} P_i \wedge o_i \neq \tau}{\text{select}\{\pi_i; P_i\}_{i \in \{1, \dots, n\}} \downarrow_{\{o_1, \dots, o_n\}}}$$

The rest is unchanged, but takes into account end actions, as well as buffer actions.

Next is extending the safety and liveness properties to channels, by adding the following definitions: (1) **Channel Safety:** A channel can be closed only once, and when closed should not be used to send a message. A closed channel can be used to receive an unbounded number of times though, and will yield a default value of the channel's type when the queue is empty; and (2) **Channel Liveness:** no channel action blocks indefinitely, ie. all channel actions lead to synchronisation on the channel eventually (or on a channel of the list of guarding actions for a select construct that has no silent guard).

► **Definition 33** (Channel Safety). Program \mathbf{P} is *channel safe* if for all P such that $\mathbf{P} \rightarrow^* (\nu \bar{u})P$, if $P \downarrow_{c^*}$ then $\neg(P \downarrow_{\text{end}[c]})$ and $\neg(P \downarrow_{\bar{c}})$.

► **Definition 34** (Channel Liveness). Program \mathbf{P} satisfies *channel liveness* if for all P such that $\mathbf{P} \rightarrow^* (\nu \bar{u})P$, (a) if $P \downarrow_c$ or $P \downarrow_{\bar{c}}$ then $P \downarrow_{\tau_c}$; and (b) if $P \downarrow_{\bar{o}}$ then $P \downarrow_{\tau_{c_i}}$ for some $c_i \in \text{fn}(\bar{o})$.

$\text{(BUF)} \frac{P \downarrow_c \quad P \downarrow_{\bar{c}} \quad \bar{v} = n}{P \mid c\langle \sigma, n \rangle :: \bar{v} \triangleright c \mapsto \bar{c}}$	$\text{(BUF-RCV)} \frac{P \downarrow_{\bar{c}} \quad \exists j \in I : \pi_j \downarrow_c \quad \bar{v} = n}{(P \mid \text{select}\{\pi_i; Q_i\}_{i \in I}) \mid c\langle \sigma, n \rangle :: \bar{v} \triangleright c \mapsto \bar{c}}$
$\text{(CL-RCV)} \frac{P \downarrow_c}{P \mid c^*\langle \sigma, n \rangle :: \bar{v} \triangleright c^* \mapsto c}$	$\text{(BUF-SND)} \frac{P \downarrow_c \quad \exists j \in I : \pi_j \downarrow_{\bar{c}} \quad \bar{v} = n}{(P \mid \text{select}\{\pi_i; Q_i\}_{i \in I}) \mid c\langle \sigma, n \rangle :: \bar{v} \triangleright c \mapsto \bar{c}}$
$\text{(SCOM)} \frac{P \downarrow_c \quad P \downarrow_{\bar{c}}}{P \mid c\langle \sigma, n \rangle :: \emptyset \triangleright \bar{c} \mapsto c}$	$\text{(SCOM-SND)} \frac{P \downarrow_c \quad \exists j \in I : \pi_j \downarrow_{\bar{c}}}{(P \mid \text{select}\{\pi_i; Q_i\}_{i \in I}) \mid c\langle \sigma, n \rangle :: \emptyset \triangleright \bar{c} \mapsto c}$
$\text{(END)} \frac{P \downarrow_{\text{end}[c]}}{P \triangleright \text{end}[c] \mapsto c^*}$	$\text{(SCOM-RCV)} \frac{P \downarrow_{\bar{c}} \quad \exists j \in I : \pi_j \downarrow_c}{(P \mid \text{select}\{\pi_i; Q_i\}_{i \in I}) \mid c\langle \sigma, n \rangle :: \emptyset \triangleright \bar{c} \mapsto c}$

We omit the symmetric rules for most rules ending in a parallel process $P \mid Q$.

■ **Figure 13** Rest of Go's Happens-Before Relation.

The channel synchronisations for the happens-before relation are listed in Figure 13. They consist of channel communication according to the official Go memory model: a send happens-before the corresponding receive, and if the channel buffer size is n , then the k -th receive happens-before the $k + n$ -th send. We add on top of that that closing a channel happens-before any default value is received from it, and when a channel is closed, default values are emitted by the closed buffer before the corresponding receive reads it.

We extend our behavioural types with the following constructs, mirroring process constructs, and using the syntax and semantics from [27, 28]:

$$S, T := \dots \mid \kappa; T \mid \text{end}[c]; T \mid \&\{\kappa_i; T_i\}_{i \in I} \mid (\nu c^n)T \mid [c]_k^n \mid c^* \quad \kappa := \bar{c} \mid c \mid \tau$$

We show the typing rules for added channel constructs, which contain the new type primitives, in Figure 14. We also add the structure rules $(\nu c)[c]_k^n \equiv \mathbf{0}$ and $(\nu c)c^* \equiv \mathbf{0}$; and the LTS semantics for the communication primitives (Figure 15). They correspond to the ones found for the processes.

$\boxed{\Gamma \vdash P \blacktriangleright T}$	$\langle \text{NEWC} \rangle \frac{\Gamma, y: \text{ch}(\sigma, n) \vdash P \blacktriangleright T \quad c \notin \text{dom}(\Gamma) \cup \text{fn}(T)}{\Gamma \vdash \text{newchan}(y: \sigma, n); P \blacktriangleright (\nu c^n)T \{c/y\}}$
$\langle \text{SND} \rangle \frac{\Gamma \vdash c: \text{ch}(\sigma, n) \quad \Gamma \vdash e: \sigma \quad \Gamma \vdash P \blacktriangleright T}{\Gamma \vdash c! \langle e \rangle; P \blacktriangleright \bar{c}; T}$	$\langle \text{RVC} \rangle \frac{\Gamma \vdash c: \text{ch}(\sigma, n) \quad \Gamma, x: \sigma \vdash P \blacktriangleright T}{\Gamma \vdash c?(x); P \blacktriangleright c; T}$
$\langle \text{BRA} \rangle \frac{\Gamma \vdash \pi_i; P_i \blacktriangleright \kappa_i; T_i}{\Gamma \vdash \text{select}\{\pi_i; P_i\}_{i \in I} \blacktriangleright \&\{\kappa_i; T_i\}_{i \in I}}$	$\langle \text{END} \rangle \frac{\Gamma \vdash P \blacktriangleright T}{\Gamma \vdash \text{close } c; P \blacktriangleright \text{end}[c]; T}$
$\boxed{\Gamma \vdash_B P \blacktriangleright T}$	$\langle \text{BUF} \rangle \frac{\Gamma \vdash c: \text{ch}(\sigma, n) \quad \bar{v} = k}{\Gamma \vdash_{\{c\}} c \langle \sigma, n \rangle :: \bar{v} \blacktriangleright [c]_k^n}$
	$\langle \text{C-BUF} \rangle \frac{\Gamma \vdash c: \text{ch}(\sigma, n)}{\Gamma \vdash_{\{c\}} c^* \langle \sigma \rangle :: \bar{v} \blacktriangleright c^*}$

■ **Figure 14** Typing Rules for Channels.

<table border="1" style="width: 100%; border-collapse: collapse;"> <tr> <td style="padding: 2px;">Channel actions</td> </tr> <tr> <td style="padding: 2px;">$\text{SND} \bar{c}; T \xrightarrow{\bar{c}} T$</td> </tr> <tr> <td style="padding: 2px;">$\text{RVC} c; T \xrightarrow{c} T$</td> </tr> <tr> <td style="padding: 2px;">$\text{END} \text{end}[c]; T \xrightarrow{\text{end}[c]} T$</td> </tr> <tr> <td style="padding: 2px;">$\text{CLD} c^* \xrightarrow{c^*} c^*$</td> </tr> <tr> <td style="padding: 2px;">$\text{BUF} [c]_k^n \xrightarrow{\text{end}[c]} c^*$</td> </tr> <tr> <td style="padding: 2px;">$k \geq 1$</td> </tr> <tr> <td style="padding: 2px;">$\text{POP} \frac{[c]_k^n \xrightarrow{c^*} [c]_{k-1}^n}{k < n}$</td> </tr> <tr> <td style="padding: 2px;">$\text{PUSH} \frac{[c]_k^n \xrightarrow{c^*} [c]_{k+1}^n}{k < n}$</td> </tr> <tr> <td style="padding: 2px;">Runtime creation</td> </tr> <tr> <td style="padding: 2px;">$\text{NEWC} (\nu c^n)T \xrightarrow{\tau} (\nu c)(T \mid [c]_0^n)$</td> </tr> </table>	Channel actions	$ \text{SND} \bar{c}; T \xrightarrow{\bar{c}} T$	$ \text{RVC} c; T \xrightarrow{c} T$	$ \text{END} \text{end}[c]; T \xrightarrow{\text{end}[c]} T$	$ \text{CLD} c^* \xrightarrow{c^*} c^*$	$ \text{BUF} [c]_k^n \xrightarrow{\text{end}[c]} c^*$	$k \geq 1$	$ \text{POP} \frac{[c]_k^n \xrightarrow{c^*} [c]_{k-1}^n}{k < n}$	$ \text{PUSH} \frac{[c]_k^n \xrightarrow{c^*} [c]_{k+1}^n}{k < n}$	Runtime creation	$ \text{NEWC} (\nu c^n)T \xrightarrow{\tau} (\nu c)(T \mid [c]_0^n)$	<table border="1" style="width: 100%; border-collapse: collapse;"> <tr> <td style="padding: 2px;">Synchronisation rules</td> </tr> <tr> <td style="padding: 2px;">$\text{CLOSE} \frac{T \xrightarrow{\text{end}[c]} T' \quad S \xrightarrow{\text{end}[c]} S'}{T \mid S \xrightarrow{\tau} T' \mid S'}$</td> </tr> <tr> <td style="padding: 2px;">$\text{SCOM} \frac{T \xrightarrow{\bar{c}} T' \quad S \xrightarrow{c} S'}{(T \mid S) \mid [c]_0^0 \xrightarrow{\tau_c} (T' \mid S') \mid [c]_0^0}$</td> </tr> <tr> <td style="padding: 2px;">$\text{OUT} \frac{T \xrightarrow{\bar{c}} T' \quad S \xrightarrow{c^*} S'}{T \mid S \xrightarrow{\tau_c} T' \mid S'}$</td> </tr> <tr> <td style="padding: 2px;">$\text{IN} \frac{T \xrightarrow{c} T' \quad S \xrightarrow{o} S' \quad o \in \{c^*, c^*\}}{T \mid S \xrightarrow{\tau_c} T' \mid S'}$</td> </tr> <tr> <td style="padding: 2px;">$\text{BRA} \frac{\kappa_j; T_j \mid T \xrightarrow{o} T' \quad o \in \{\tau, \tau_c\}}{\&\{\kappa_i; T_i\}_{i \in I} \mid T \xrightarrow{o} T'}$</td> </tr> </table>	Synchronisation rules	$ \text{CLOSE} \frac{T \xrightarrow{\text{end}[c]} T' \quad S \xrightarrow{\text{end}[c]} S'}{T \mid S \xrightarrow{\tau} T' \mid S'}$	$ \text{SCOM} \frac{T \xrightarrow{\bar{c}} T' \quad S \xrightarrow{c} S'}{(T \mid S) \mid [c]_0^0 \xrightarrow{\tau_c} (T' \mid S') \mid [c]_0^0}$	$ \text{OUT} \frac{T \xrightarrow{\bar{c}} T' \quad S \xrightarrow{c^*} S'}{T \mid S \xrightarrow{\tau_c} T' \mid S'}$	$ \text{IN} \frac{T \xrightarrow{c} T' \quad S \xrightarrow{o} S' \quad o \in \{c^*, c^*\}}{T \mid S \xrightarrow{\tau_c} T' \mid S'}$	$ \text{BRA} \frac{\kappa_j; T_j \mid T \xrightarrow{o} T' \quad o \in \{\tau, \tau_c\}}{\&\{\kappa_i; T_i\}_{i \in I} \mid T \xrightarrow{o} T'}$
Channel actions																		
$ \text{SND} \bar{c}; T \xrightarrow{\bar{c}} T$																		
$ \text{RVC} c; T \xrightarrow{c} T$																		
$ \text{END} \text{end}[c]; T \xrightarrow{\text{end}[c]} T$																		
$ \text{CLD} c^* \xrightarrow{c^*} c^*$																		
$ \text{BUF} [c]_k^n \xrightarrow{\text{end}[c]} c^*$																		
$k \geq 1$																		
$ \text{POP} \frac{[c]_k^n \xrightarrow{c^*} [c]_{k-1}^n}{k < n}$																		
$ \text{PUSH} \frac{[c]_k^n \xrightarrow{c^*} [c]_{k+1}^n}{k < n}$																		
Runtime creation																		
$ \text{NEWC} (\nu c^n)T \xrightarrow{\tau} (\nu c)(T \mid [c]_0^n)$																		
Synchronisation rules																		
$ \text{CLOSE} \frac{T \xrightarrow{\text{end}[c]} T' \quad S \xrightarrow{\text{end}[c]} S'}{T \mid S \xrightarrow{\tau} T' \mid S'}$																		
$ \text{SCOM} \frac{T \xrightarrow{\bar{c}} T' \quad S \xrightarrow{c} S'}{(T \mid S) \mid [c]_0^0 \xrightarrow{\tau_c} (T' \mid S') \mid [c]_0^0}$																		
$ \text{OUT} \frac{T \xrightarrow{\bar{c}} T' \quad S \xrightarrow{c^*} S'}{T \mid S \xrightarrow{\tau_c} T' \mid S'}$																		
$ \text{IN} \frac{T \xrightarrow{c} T' \quad S \xrightarrow{o} S' \quad o \in \{c^*, c^*\}}{T \mid S \xrightarrow{\tau_c} T' \mid S'}$																		
$ \text{BRA} \frac{\kappa_j; T_j \mid T \xrightarrow{o} T' \quad o \in \{\tau, \tau_c\}}{\&\{\kappa_i; T_i\}_{i \in I} \mid T \xrightarrow{o} T'}$																		

■ **Figure 15** Remaining LTS Semantics of Types.

All results in § 5 hold as-is with the new definitions. We only add the new barbs, like for processes (identical definition), and the following type properties:

► **Definition 35** (Channel Safety). Type T is *channel safe* if for all T such that $T \rightarrow^* (\nu \tilde{u})T$, if $T \downarrow_{c^*}$ then $\neg(T \downarrow_{\text{end}[c]})$ and $\neg(T \downarrow_{\bar{c}})$.

► **Definition 36** (Channel Liveness). Type T is *channel live* if for all T such that $T \rightarrow^* (\nu \tilde{u})T$, (a) if $T \downarrow_c$ or $T \downarrow_{\bar{c}}$ then $T \downarrow_{\tau_c}$; and (b) if $T \downarrow_{\bar{o}}$ then $T \downarrow_{\tau_{c_i}}$ for some $c_i \in \text{fn}(\bar{o})$.

They correspond to the ones added for processes, and are integrated in other theorems of § 5.

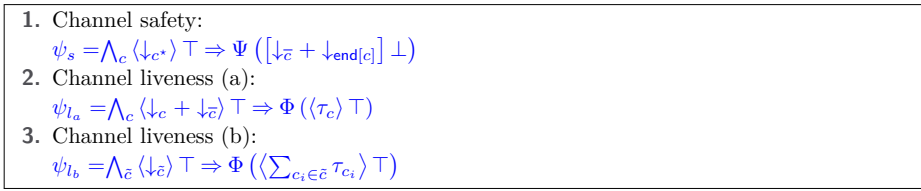
7.3 Modal μ -Calculus Properties for Channels

With extending to the channel primitives, all definitions in § 6 still hold with added properties in the modal μ -calculus for channel liveness and safety. These are defined in Figure 16.

The model-checking result is also extended as the following theorem to capture the situation where shared memory and message passing co-exist.

► **Theorem 37** (Model Checking of GoL processes). *Suppose $\Gamma \vdash P \blacktriangleright T$.*

1. If $T \models_{\tau_P} \Psi(\phi)$ for $\phi \in \{\psi_{s_a}, \psi_{s_b}, \psi_s, \psi_d\}$, then $P \models_{\tau_P} \Psi(\phi)$.
2. If $T \models_{\tau_P} \Psi(\phi)$ for $\phi \in \{\psi_l, \psi_{l_a}, \psi_{l_b}\}$ and either (a) $P \in \text{May}\downarrow$ or (b) $P \notin \text{Inf}$ or (c) $P \in \text{AC}$, then $P \models_{\tau_P} \Psi(\phi)$.



■ **Figure 16** Modal μ -calculus properties for channels.

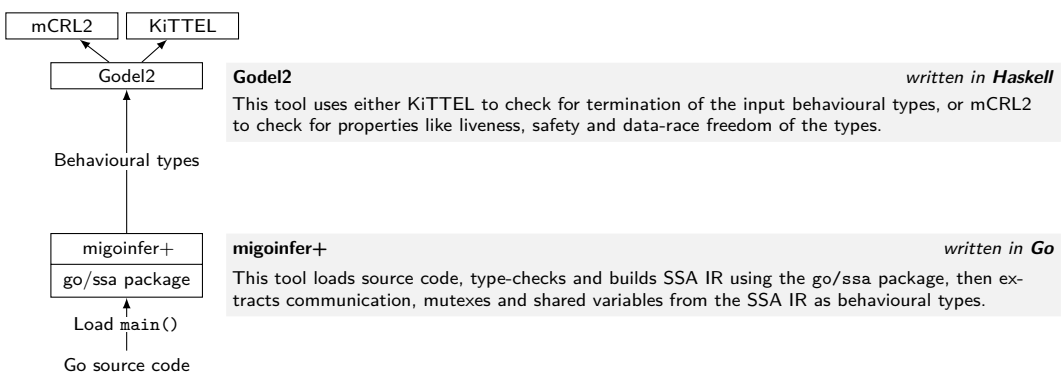
This extension to our framework allows us not only to integrate the previous framework by [27, 28], but also show to some extent the modularity of our memory-based approach. With channels, this extension of GoL is implementing a significant range of the concurrency features of Go, allowing for a range of programs to be model-checked for data races, liveness issues and other safety issues in the use of locks and channels.

7.4 Types and process (program) liveness

There are several categories of processes for which the equivalence between types and process (program) liveness is not ensured: (3) programs that have an infinite conditional that is not an alternating conditional, *if they do not always have a termination path available*. They can be checked by the model checker if they are not in (3), however the result may not coincide with the process liveness; (2) programs that neither have an infinite conditional, nor always have a potential path for termination (e.g. a program that recurses indefinitely without ever having an ending branch available through a `select` construct, without the need of a conditional in the recursing selection); and (3) programs that are not finite control – i.e. programs that spawn an unbounded amount of new processes – because the model-checker will not be able to generate a linear representation of them (see § 8).

Note that for (1) and (2), the tool returns “live” if the types are live, though it may be the case that the programs are not live.

8 Implementation and Evaluation



■ **Figure 17** Workflow of the verification toolchain.

The tool chain. Our implementation tool (shown in Figure 17) consists of a type inference tool and a type verifier. The type inference tool (`migoinfer+`) [3] extracts behavioural types, including eight new primitives related to shared memory: creating a new lock (called mutex

in the tool, in reference to the name of the mutual exclusion lock implementation in Go) or shared address, exclusive write-locking or unlocking of a lock or a read-write lock, read-locking/unlocking a read-write lock, and reading or writing a shared variable. This new inference tool supports both channel-based communication primitives from [28] and shared memory primitives.

`migoinfer+` currently supports a subset of the Go language syntax, extracting only variables and mutexes created explicitly inside the body of a function, and does not support embedding or mutexes in `struct`. These usage patterns of mutexes can be transformed to the flat representation we support, allowing us to analyse the examples in our benchmark [1]. Note that it is advised to avoid the non-declared sharing of variables, channels and mutexes to a nameless child goroutine, as it may not extract the parameter passing properly, and this is a good practice in Go to specify shared parameters. Programs that spawn an unbounded number of goroutines such as our prime-sieve example can be extracted by `migoinfer+` if they respect the above limitations. Lastly, the use of some (non-default) packages, such as the `net` package, is known to break `migoinfer+` under certain conditions, making it not extract the types correctly.

The type verifier (Godel2) [2] analyses the new extracted primitives, implements the theory presented in this paper, and uses the mCRL2 [44, 20] model checker as a backend to check safety and data race properties. Regarding the liveness properties, as discussed after Theorem 16 and in [27, 28], liveness of types does not imply liveness of processes, due to conditionals behaving differently in the types and the processes. In Theorem 30, we identified the three classes of Go programs where both liveness properties coincide. One such class is a set of terminating processes, as defined in Definition 23, which is a strict subset of may converging processes (Proposition 24). To make sure liveness coincides on types and processes, we combine the termination checker KITTeL [11] to our tool (see also [28, § 5]). This tool can check processes that are not terminating under certain conditions, namely they should not spawn an unbounded number of threads. However, such programs may, in rare cases, lead to false positives or negatives regarding liveness (and possibly safety), because of the approximations the model checker has to make when running against models with cycles.

Evaluations. We evaluate our tool for reference on an 8-core Intel i7-7700K machine with 16 GB memory, in a 64-bit Linux environment running go 1.12.2. Table 1 shows the results for a range of programs that mix shared memory with either channels or mutexes as locking mechanism. The sources for those examples can be found in the benchmark repository [1]. Programs `no-race` and `simple-race` are programs made to test the behaviour of mutexes and check that liveness errors are properly reported. The channel version of our running example, from Figure 11 is named `channel-as-lock`, and `channel-as-lock-bad` is a variation of the `-fixed` version but with channel sends and receive switched, hence the program deadlocks on the first attempt to lock of each thread as there is nothing to receive.

The `deposit` implementation is taken from [10] (the example to present data races and locking mechanisms), and `prod-cons` is a shared memory implementation of the classic producer-consumer algorithm, where two producers race against each other and one consumer takes whichever product is available first. In this example, all three threads share a single memory heap, supposed to be protected by a mutex. Finally, `dine5` is an implementation of the Dining Philosophers problem as explained in § 5.3, and `dine5-chan` is a channel variant adapted slightly to allow for a potential shared-memory data race.

We note that the Prime Sieve algorithm [27, 33] is not analysed by our tool, as it continually spawns new threads, making the state space too big for the mCRL2 model-checker.

■ **Table 1** Go Programs Verified by the Toolchain.

Programs	LoC	Sum	Safe	Live	DRF	time (ms)
no-race	15	9	✓	✓	✓	691.45
no-race-mutex	24	33	✓	✓	✓	785.57
no-race-mut-bad	23	20	✓	✗	✓	721.77
simple-race	13	8	✓	✓	✗	701.93
simple-race-fix	19	17	✓	✓	✓	731.73
deposit-race ¹	18	14	✓	✓	✗	697.90
deposit-fix ¹	24	27	✓	✓	✓	727.43
ch-as-lock-race ²	19	20	✓	✓	✗	753.99
ch-as-lock-fix ²	19	20	✓	✓	✓	745.64
ch-as-lock-bad	19	20	✓	✗	✓	749.97
prod-cons-race	38	156	✓	✓	✗	1,903.52
prod-cons-fix	40	188	✓	✓	✓	1,971.26
dine5-unsafe	35	106	✗	✓	✓	6,996.27
dine5-deadlock	35	106	✓	✗	✓	12,278.33
dine5-fix	35	106	✓	✓	✓	8,998.04
dine5-chan-race	59	2672	✓	✓	✗	~ 185mn
dine5-chan-fix	59	2688	✓	✓	✓	~ 645mn

¹[10], ²Figure 11, LoC: Lines of Code, DRF: Data Race Free, Sum: Summands, ✓: Formula is true, ✗: Formula is false

Future work for applying this approach to real-world Go programs are: working around the explosion seen with `select+channels` in `dine5-chan`, for which using a different model for `select` constructs and channel actions than the one in our implementation might be sufficient; working on the implementation for a wider range of extractions for channels, shared memory and mutexes embedded in `structs`, or to implement a parser that flattens those `structs` upstream of `migoinfer+`; and working on analysis of programs that dynamically spawn new goroutines – this would require non-trivial approximations to be leveraged. Note that it should represent only a small fraction of programs, as most daily-use protocols should be implementable without the need for such unbounded growth in memory usage.

All examples in Table 1 are analysed by our tool, and the time given as an indication scales exponentially with the number of summands (and possibly action labels) and their ordering, in the linear process specification that represents the types in the model checker. Those directly depend from the source code of the analysed program.

9 Conclusion and Related Work

The Go language provides a unique programming environment where both explicit communication and shared memory concurrency primitives co-exist. This work introduces `GoL` as an abstraction layer for Go code, as well as behavioural types to propose a static verification framework for detecting concurrency bugs in Go. These include deadlocks and safety for both mutual exclusion locks and channel communication, as well as data race detection for shared memory primitives.

Shared memory locks and channels cover by themselves a substantial amount of Go’s concurrency features. The former is a low-level, standard library provision and the latter is a high-level, built-in language feature. Go only features these two basic building blocks because one can use them to implement most higher levels of concurrency abstraction, for example actors models.

The works [27, 28] built behavioural types for verification of concurrency bugs for channel-based message passing. We integrate with their asynchronous calculus (a.k.a. `AMiGo`) for our channel-related extension in § 7. These works, however, were lacking more shared memory

concurrency with locks and shared pointers, and did not tackle data races for shared pointers, which we do. It does not study happens before relations either (for channels). It furthermore was lacking complete proofs on their equivalence theorems for liveness, which is also addressed in this paper. We also proved GoL satisfies the properties of the types characterised by the modal μ -calculus (Theorems 30,37). The paper [28] has informally described them, but these have never been formalised nor proved.

The work [42] defines forkable behaviours (ie. regular expressions with a fork construct) to capture goroutine spawning in synchronous Go programs. They develop a tool based on this model to analyse directly Go programs. Their approach is sound, but suffers from several limitations, which were overcome by [27, 28]; their tool does not treat shared memory concurrency primitives and locks.

The work [25] observed that asynchronous distributed systems can be verified by only modelling *synchronisations* in the core protocol, and introduces a language IceT similar to GoL for specifying synchronisation in message-passing programs. Their focus was to verify functional correctness of the input protocol, and requires input programs to be synchronisable (i.e. no deadlocks nor spurious sends in the input programs). Their approach allows for checking correctness of an implementation, given a reasonable amount of annotations. It is orthogonal to our work in which we only need to check for runtime sanity. Both approaches independently benefit the user, and should be run individually on testing code in order to check both for concurrency behavioural bugs and for implementation bugs.

Recent works [46, 9] provide empirical studies of Go programs, which show that almost half of concurrency bugs in Go are non-blocking bugs, mostly shared memory problems, and the remaining blocking bugs are mostly related to channel and lock misuse. That gives an incentive to make tools and implementations built on the concurrent behavioural theory, for easy detection of such bugs. Our work is part of that effort.

A large body of race detection tools targeting other languages such as Java are available. ThreadSanitizer (TSan) [40, 45, 41] which is included in LLVM/Clang is one of the most widely deployed dynamic race detectors. The runtime race detector of Go [15] uses TSan's runtime library.

The work [30] proposes a subset of the Go language akin to GoL, along with a modular approach to statically analyse processes. Their approach combines lattice-valued regular expressions and a shuffle operator allowing for separate analysis of single threads, and they prove their theory to be sound. They have a prototype implementation in OCaml to check deadlocks in synchronous message-passing programs. The work [6] uses a protocol description language, Scribble [39], which is a practical incarnation of multiparty session types [23] to generate Go APIs, ensuring deadlock freedom and liveness of communications by construction. Neither [30] nor [6] treat either communication error or data race detection, both handled in this paper, nor do they treat shared variables, which our approach extends upon.

The main difference in code writing between Go and GoL is the handling of continuations for select and if-then-else constructs, where Go allows for standard continuation while GoL restrains the user to use tail calls. This is handled by our extraction tool, as it extracts the Go code to GoL by building an SSA representation before extracting relevant primitives from it, see Figure 17 in § 8.

The idea to use the LTS of behavioural types for programming analysis dates back to [34] for Concurrent ML, and since then, it has been applied to many works [5]. Some tackle mutual exclusion locks, but systematically lack support for read-write mutual exclusion locks, including works [24, 4, 21]. The work [26] aims to guarantee liveness with termination of a typed π -calculus. We study wider classes in the theory, aiming termination to use the existing tool (KITTeL) in order to integrate with our tool-chain to scale – thus the main aim and the target (real Go programs in our case) differ from [26].

Type-level model-checking for message-passing programming was first addressed in [7]. Recent applications using mCRL2 include verifications of multiparty session typed π -calculus [37] and the Dotty programming language (the future Scala 3) [38].

Our future works include studying the soundness and completeness of the happens-before relation provided by the Go memory model, ie. studying if the definition of data race given by it covers all data races that can happen in Go, and whether it does not provide false positives; speeding-up the analysis using more mCRL2 options and the extension to an incremental analysis based on happens-before relations, as taken in other languages, e.g. [29, 49]; as well as possibly counter-example extraction for code failing verification, to provide direct access to the detected bugs to developers. There is also the possibility to work on handling dynamic process creation, widening the analysis scope of our current tool and model.

References

- 1 Godel 2 Benchmarks. URL: <https://github.com/JujuYuki/godel2-benchmark>.
- 2 Godel 2. URL: <https://github.com/JujuYuki/godel2>.
- 3 migoinfer+. URL: <https://github.com/JujuYuki/gospal>.
- 4 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.
- 5 Davide Ancona, Viviana Bono, Mario Bravetti, Joana Campos, Giuseppe Castagna, Pierre-Malo Deniérou, Simon J. Gay, Nils Gesbert, Elena Giachino, Raymond Hu, Einar Broch Johnsen, Francisco Martins, Viviana Mascardi, Fabrizio Montesi, Rumyana Neykova, Nicholas Ng, Luca Padovani, Vasco T. Vasconcelos, and Nobuko Yoshida. Behavioral Types in Programming Languages. *Foundations and Trends in Programming Languages*, 3(2-3), 2017. doi:10.1561/25000000031.
- 6 David Castro, Raymond Hu, Sung-Shik Jongmans, Nicholas Ng, and Nobuko Yoshida. Distributed Programming Using Role Parametric Session Types in Go. In *46th ACM SIGPLAN Symposium on Principles of Programming Languages*, pages 1–30. ACM, 2019. doi:10.1145/3290342.
- 7 Sagar Chaki, Sriram K. Rajamani, and Jakob Rehof. Types as models: model checking message-passing programs. In *POPL’02*, pages 45–57, 2002.
- 8 Sjoerd Cranen, Jan Friso Groote, Jeroen J. A. Keiren, Frank P. M. Stappers, Erik P. de Vink, Wieger Wesselink, and Tim A. C. Willemse. *An Overview of the mCRL2 Toolset and Its Recent Advances*, pages 199–213. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013. doi:10.1007/978-3-642-36742-7_15.
- 9 Nicolas Dille and Julien Lange. An empirical study of messaging passing concurrency in Go projects. In *SANER*. IEEE, 2019.
- 10 Alan A.A. Donovan and Brian W. Kernighan. *The Go Programming Language*. Addison-Wesley Professional, 1st edition, 2015.
- 11 Stephan Falke and Marc Brockschmidt. KITTeL/KoAT. <https://github.com/s-falke/kittel-koat>, 2018.
- 12 Steve Francia. Nine years of Go. <https://blog.golang.org/9years>, 2018.
- 13 Julia Gabet and Nobuko Yoshida. Static Race Detection and Mutex Safety and Liveness for Go Programs (extended version). *CoRR*, abs/2004.12859, 2020. arXiv:2004.12859.
- 14 GitHub. The fastest growing languages, 2018. URL: <http://octoverse.github.com/>.
- 15 Go. Data Race Detector. https://golang.org/doc/articles/race_detector.html, 2013.
- 16 Go. The Go Memory Model. <https://golang.org/ref/mem>, 2014.
- 17 Golang. mutex.go, 2019. URL: <https://golang.org/src/sync/mutex.go>.
- 18 Golang. rwmutex.go, 2019. URL: <https://golang.org/src/sync/rwmutex.go>.
- 19 Golang. The Go Programming Language, 2019. URL: <https://golang.org>.

- 20 Jan Friso Groote and Mohammad Reza Mousavi. *Modeling and Analysis of Communicating Systems*. The MIT Press, 2014.
- 21 Jonas Kastberg Hinrichsen, Jesper Bengtson, and Robbert Krebbers. Actris: Session-type based reasoning in separation logic. *Proc. ACM Program. Lang.*, 4(POPL), December 2019. doi:10.1145/3371074.
- 22 C.A.R. Hoare. *Communicating Sequential Processes*. Prentice Hall, 1985.
- 23 Kohei Honda, Nobuko Yoshida, and Marco Carbone. Multiparty asynchronous session types. In *POPL'08*, pages 273–284. ACM, 2008.
- 24 Atsushi Igarashi and Naoki Kobayashi. A generic type system for the pi-calculus. In *Proceedings of the 28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '01, pages 128–141, New York, NY, USA, 2001. ACM. doi:10.1145/360204.360215.
- 25 Klaus v. Gleissenthall and Rami Gökhan Kıcı and Alexander Bakst and Deian Stefan and Ranjit Jhala. Pretend synchrony: Synchronous verification of asynchronous distributed programs. *Proc. ACM Program. Lang.*, 3(POPL):59:1–59:30, January 2019. doi:10.1145/3290372.
- 26 Naoki Kobayashi and Davide Sangiorgi. A hybrid type system for lock-freedom of mobile processes. *ACM Trans. Program. Lang. Syst.*, 32(5), May 2008. doi:10.1145/1745312.1745313.
- 27 Julien Lange, Nicholas Ng, Bernardo Toninho, and Nobuko Yoshida. Fencing off Go: Liveness and safety for channel-based programming. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*, POPL 2017, pages 748–761. ACM, 2017. doi:10.1145/3009837.3009847.
- 28 Julien Lange, Nicholas Ng, Bernardo Toninho, and Nobuko Yoshida. A static verification framework for message passing in go using behavioural types. In *Proceedings of the 40th International Conference on Software Engineering*, ICSE '18, pages 1137–1148, New York, NY, USA, 2018. ACM. doi:10.1145/3180155.3180157.
- 29 Bozhen Liu and Jeff Huang. D4: Fast concurrency debugging with parallel differential analysis. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2018, pages 359–373, New York, NY, USA, 2018. ACM. doi:10.1145/3192366.3192390.
- 30 Jan Midtgaard, Flemming Nielson, and Hanne Riis Nielson. Process-local static analysis of synchronous processes. In Andreas Podelski, editor, *Static Analysis*, pages 284–305, Cham, 2018. Springer International Publishing.
- 31 Robin Milner. *A Calculus of Communicating Systems*, volume 92. Springer-Verlag, 1980.
- 32 Robin Milner and Davide Sangiorgi. Barbed bisimulation. In *Proc. ICALP'92*, volume 623 of *LNCS*, 1992.
- 33 Nicholas Ng and Nobuko Yoshida. Static Deadlock Detection for Concurrent Go by Global Session Graph Synthesis. In *CC 2016*, pages 174–184. ACM, 2016.
- 34 Hanne Riis Nielson and Flemming Nielson. Higher-order concurrent programs with finite communication topology (extended abstract). In *POPL*, 1994. doi:10.1145/174675.174538.
- 35 Rob Pike. Go at Google. In *SPLASH*, pages 5–6, New York, NY, USA, 2012. ACM.
- 36 Davide Sangiorgi and David Walker. *The π -Calculus: a Theory of Mobile Processes*. Cambridge University Press, 2001.
- 37 Alceste Scalas and Nobuko Yoshida. Less Is More: Multiparty Session Types Revisited. In *46th ACM SIGPLAN Symposium on Principles of Programming Languages*, pages 1–29. ACM, 2019.
- 38 Alceste Scalas, Nobuko Yoshida, and Elias Benussi. Verifying message-passing programs with dependent behavioural types. In *Programming Language Design and Implementation*, 2019.
- 39 Scribble. Scribble Project, 2008. URL: www.scribble.org.
- 40 Konstantin Serebryany and Timur Iskhodzhanov. ThreadSanitizer: Data race detection in practice. In *Proceedings of the Workshop on Binary Instrumentation and Applications*, WBIA '09, pages 62–71, New York, NY, USA, 2009. ACM. doi:10.1145/1791194.1791203.

- 41 Konstantin Serebryany, Alexander Potapenko, Timur Iskhodzhanov, and Dmitriy Vyukov. Dynamic race detection with LLVM compiler. In *Proceedings of the Second International Conference on Runtime Verification, RV'11*, pages 110–114, Berlin, Heidelberg, 2012. Springer-Verlag. doi:10.1007/978-3-642-29860-8_9.
- 42 Kai Stadtmüller, Martin Sulzmann, and Peter Thiemann. Static Trace-Based Deadlock Analysis for Synchronous Mini-Go. In *APLAS*, volume 10017 of *LNCS*, 2016.
- 43 Syzkaller. Randomized testing for Go. <https://github.com/google/syzkaller>, 2015.
- 44 Technische Universiteit Eindhoven. mCRL2. https://www.mcrl2.org/web/user_manual/index.html, 2018.
- 45 The Clang Team. ThreadSanitizer. <http://clang.llvm.org/docs/ThreadSanitizer.html>, 2015.
- 46 Tengfei Tu, Xiaoyu Liu, Linhai Song, and Yiyang Zhang. Understanding real-world concurrency bugs in Go. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '19*, pages 865–878, New York, NY, USA, 2019. ACM. doi:10.1145/3297858.3304069.
- 47 Dmitry Vyukov. Randomized testing for Go. <https://github.com/dvyukov/go-fuzz>, 2015.
- 48 Dmitry Vyukov and Andrew Gerrand. Introducing the Go Race Detector. <https://blog.golang.org/race-detector>, 2013.
- 49 Sheng Zhan and Jeff Huang. Echo: Instantaneous in situ race detection in the IDE. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2016*, pages 775–786, New York, NY, USA, 2016. ACM. doi:10.1145/2950290.2950332.

Reconciling Event Structures with Modern Multiprocessors

Evgenii Moiseenko

St. Petersburg State University, Russia
JetBrains Research, St. Petersburg, Russia
e.moiseenko@2012.spbu.ru

Anton Podkopaev

National Research University Higher School of Economics, Moscow, Russia
MPI-SWS, Kaiserslautern, Germany
JetBrains Research, St. Petersburg, Russia
podkopaev@mpi-sws.org

Ori Lahav

Tel Aviv University, Israel
orilahav@tau.ac.il

Orestis Melkonian

University of Edinburgh, UK
melkon.or@gmail.com

Viktor Vafeiadis

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

Abstract

Weakestmo is a recently proposed memory consistency model that uses event structures to resolve the infamous “out-of-thin-air” problem and to enable efficient compilation to hardware. Nevertheless, this latter property – compilation correctness – has not yet been formally established.

This paper closes this gap by establishing correctness of the intended compilation schemes from **Weakestmo** to a wide range of formal hardware memory models (x86, POWER, ARMv7, ARMv8) in the Coq proof assistant. Our proof is the first that establishes correctness of compilation of an event-structure-based model that forbids “out-of-thin-air” behaviors, as well as the first mechanized compilation proof of a weak memory model supporting sequentially consistent accesses to such a range of hardware platforms. Our compilation proof goes via the recent Intermediate Memory Model (IMM), which we suitably extend with sequentially consistent accesses.

2012 ACM Subject Classification Theory of computation → Logic and verification; Software and its engineering → Concurrent programming languages

Keywords and phrases Weak Memory Consistency, Event Structures, IMM, Weakestmo

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2020.5

Related Version A full version of the paper is available at <http://plv.mpi-sws.org/weakestmoToImm/>.

Supplementary Material ECOOP 2020 Artifact Evaluation approved artifact available at <https://doi.org/10.4230/DARTS.6.2.4>.

Funding *Evgenii Moiseenko*: was supported by RFBR (grant number 18-01-00380).

Anton Podkopaev: was supported by RFBR (grant number 18-01-00380).

Ori Lahav: was supported by the Israel Science Foundation (grant number 5166651), by Len Blavatnik and the Blavatnik Family foundation, and by the Alon Young Faculty Fellowship.



© Evgenii Moiseenko, Anton Podkopaev, Ori Lahav, Orestis Melkonian, and Viktor Vafeiadis;
licensed under Creative Commons License CC-BY

34th European Conference on Object-Oriented Programming (ECOOP 2020).

Editors: Robert Hirschfeld and Tobias Pape; Article No. 5; pp. 5:1–5:26

Leibniz International Proceedings in Informatics



Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany



1 Introduction

A major research problem in concurrency semantics is to develop a weak memory model that allows load-to-store reordering (a.k.a. *load buffering*, LB) and compiler optimizations (e.g., elimination of fake dependencies), while forbidding “out-of-thin-air” behaviors [18, 10, 5, 13].

The problem can be illustrated with the following two programs, which access locations x and y initialized to 0. The annotated outcome $a = b = 1$ ought to be allowed for LB-fake because $1 + a * 0$ can be optimized to 1 and then the instructions of thread 1 executed out of order. In contrast, it should be forbidden for LB-data, since no optimizations are applicable.

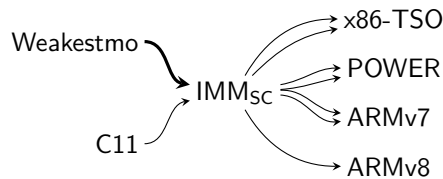
$$\begin{array}{c}
 a := [x] \ //1 \\
 [y] := 1 + a * 0
 \end{array}
 \parallel
 \begin{array}{c}
 b := [y] \ //1 \\
 [x] := b
 \end{array}
 \quad (\text{LB-fake})
 \quad \Bigg| \quad
 \begin{array}{c}
 a := [x] \ //1 \\
 [y] := a
 \end{array}
 \parallel
 \begin{array}{c}
 b := [y] \ //1 \\
 [x] := b
 \end{array}
 \quad (\text{LB-data})$$

Among the proposed models that correctly distinguish between these two programs is the recent *Weakestmo* model [6]. *Weakestmo* was developed in response to certain limitations of earlier models, such as the “promising semantics” of Kang et al. [11], namely that (i) they did not cover the whole range of C/C++ concurrency features and that (ii) they did not support the intended compilation schemes to hardware.

Being flexible in its design, *Weakestmo* addresses the former point. It supports all usual features of the C/C++11 model [3] and can easily be adapted to support any new concurrency features that may be added in the future. It does not, however, fully address the latter point. Due to the difficulty of establishing correctness of the intended compilation schemes to hardware architectures that permit load-store reordering (i.e., POWER, ARMv7, ARMv8), Chakraborty and Vafeiadis [6] only establish correctness of suboptimal schemes that add (unnecessary) explicit fences to prevent load-store reordering.

In this paper, we address this major limitation of the *Weakestmo* paper. We establish in Coq correctness of the intended compilation schemes to a wide range of hardware architectures that includes the major ones: x86-TSO [17], POWER [1], ARMv7 [1], ARMv8 [21]. The compilation schemes, whose correctness we prove, do not require any fences or fake dependencies for relaxed accesses. Because of a technical limitation of our setup (see §6), however, compilation of read-modify-write (RMW) accesses to ARMv8 uses a load-reserve/store-conditional loop (similar to that of ARMv7 and POWER) as opposed to the newly introduced ARMv8 instructions for certain kinds of RMWs.

The main challenge in this proof is to reconcile the different ways in which hardware models and *Weakestmo* allow load-store reordering. Unlike most models at the programming language level, hardware models (such as ARMv8) do not execute instructions in sequence; they instead keep track of dependencies between instructions and ensure that no dependency cycles ever arise in a single execution. In contrast, *Weakestmo* executes instructions in order, but simultaneously considers multiple executions to justify an execution where a load reads a value that indirectly depends upon a later store. Technically, these multiple executions together form an *event structure*, upon which *Weakestmo* places various constraints.



■ **Figure 1** Results proved in this paper.

The high-level proof structure is shown in Fig. 1. We reuse IMM, an *intermediate memory model*, introduced by Podkopaev et al. [19] as an abstraction over all major existing hardware memory models. To support Weakestmo compilation, we extend IMM with *sequentially consistent* (SC) accesses following the RC11 model [13]. As IMM is very much a hardware-like model (e.g., it tracks dependencies), the main result is compilation from Weakestmo to IMM (indicated by the bold arrow). The other arrows in the figure are extensions of previous results to account for SC accesses, while double arrows indicate results for two compilation schemes.

The complexity of the proof is also evident from the size of the Coq development. We have written about 30K lines of Coq definitions and proof scripts on top of an existing infrastructure of about another 20K lines (defining IMM, the aforementioned hardware models and many lemmas about them). As part of developing the proof, we also had to mechanize the Weakestmo definition in Coq and to fix some minor deficiencies in the original definition, which were revealed by our proof effort.

To the best of our knowledge, our proof is the first proof of correctness of compilation of an event-structure-based memory model. It is also the first mechanized compilation proof of a weak memory model supporting sequentially consistent accesses to such a range of hardware architectures. The latter, although fairly straightforward in our case, has had a history of wrong compilation correctness arguments (see [13] for details).

Outline. We start with an informal overview of IMM, Weakestmo, and our compilation proof (§2). We then present a fragment of Weakestmo formally (§3) and its compilation proof (§4). Subsequently, we extend these results to cover SC accesses (§5), discuss related work (§6) and conclude (§7). The associated proof scripts and supplementary material for our paper are publicly available at <http://plv.mpi-sws.org/weakestmoToImm/>.

2 Overview of the Compilation Correctness Proof

To get an idea about the IMM and Weakestmo memory models, consider a version of the LB-fake and LB-data programs from §1 with no dependency in thread 1:

$$\begin{array}{l} a := [x] \ // 1 \\ [y] := 1 \end{array} \parallel \begin{array}{l} b := [y] \ // 1 \\ [x] := b \end{array} \quad (\text{LB})$$

As we will see, the annotated outcome is allowed by both IMM and Weakestmo, albeit in different ways. The different treatment of load-store reordering affects the outcomes of other programs. For example, IMM forbids the annotated outcome of LB-fake by treating it exactly as LB-data, whereas Weakestmo allows the outcome by treating LB-fake exactly as LB.

2.1 An Informal Introduction to IMM

IMM is a *declarative* (also called *axiomatic*) model identifying a program’s semantics with a set of *execution graphs*, or just *executions*. As an example, Fig. 2a contains G_{LB} , an IMM execution graph of LB corresponding to an execution yielding the annotated behavior.

Vertices of execution graphs, called *events*, represent memory accesses either due to the initialization of memory or to the execution of program instructions. Each event is labeled with the type of the access (e.g., R for reads, W for writes), the location accessed, and the value read or written. Memory initialization consists of a set of events labeled $W(x, 0)$ for each location x used in the program; for conciseness, however, we depict the initialization events as a single event with label *init*.

(a) G_{LB} : Execution graph of LB.

(b) Execution of LB-data and LB-fake.

■ **Figure 2** Executions of LB and LB-data/LB-fake with outcome $a = b = 1$.

Edges of execution graphs represent different relations on events. In Fig. 2, three different relations are depicted. The *program order* relation (po) totally orders events originated from the same thread according to their order in the program, as well as the initialization event(s) before all other events. The *reads-from* relation (rf) relates a write event to the read events that read from it. Finally, the *preserved program order* (ppo) is a subset of the program order relating events that cannot be executed out of order. Such ppo edges arise whenever there is a dependency chain between the corresponding instructions (e.g., a write storing the value read by a prior read).

Because of the syntactic nature of ppo , IMM conflates the executions of LB-data and LB-fake leading to the outcome $a = b = 1$ (see Fig. 2b). This choice is in line with hardware memory models; it means, however, that IMM is not suitable as a memory model for a programming language (because, as argued in §1, LB-fake can be transformed to LB by an optimizing compiler).

The executions of a program are constructed in two steps.¹ First, a thread-local semantics determines the sequential executions of each thread, where the values returned by each read access are chosen non-deterministically (among the set of *all* possible values), and the executions of different threads are combined into a single execution. Then, the execution graphs are filtered by a *consistency predicate*, which determines which executions are allowed (i.e., are IMM-consistent). These IMM-consistent executions form the program’s semantics.

IMM-consistency checks three basic constraints:

Completeness: Every read event reads from precisely one write with the same location and value;

Coherence: For each location x , there is a total ordering of x -related events extending the program order so that each read of x reads from the most recent prior write according to that total order; and

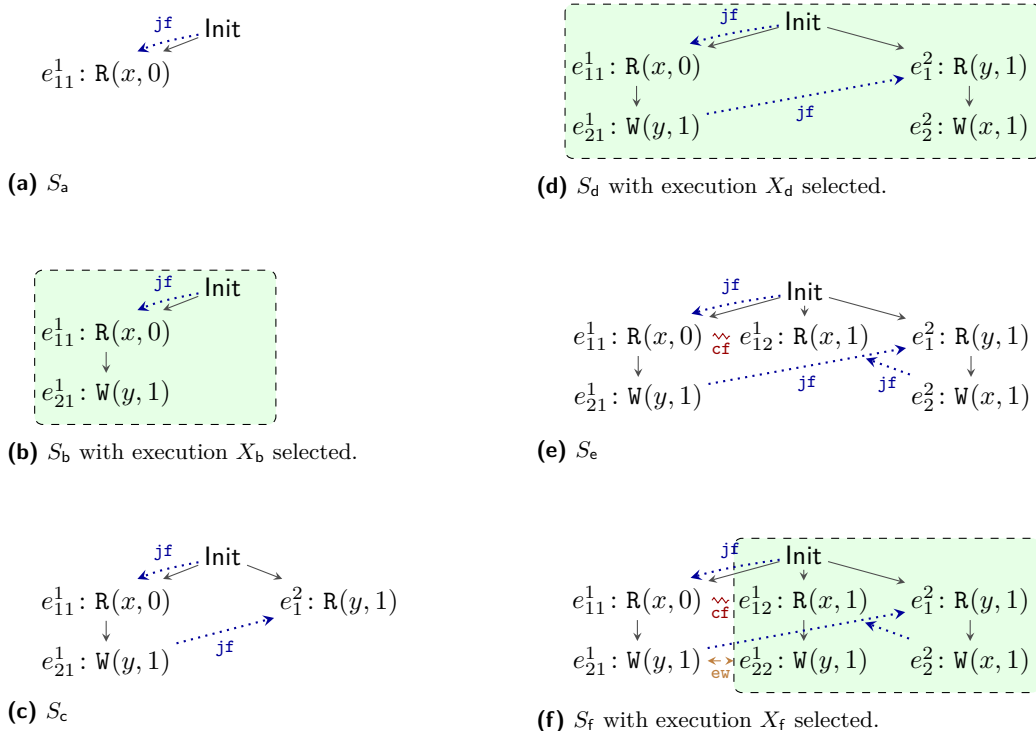
Acyclic dependency: There is no cycle consisting only of ppo and rf edges.

The final constraint disallows executions in which an event recursively depends upon itself, as this pattern can lead to “out-of-thin-air” outcomes. Specifically, the execution in Fig. 2b, which represents the annotated behavior of LB-fake and LB-data, is *not* IMM-consistent because of the $(ppo \cup rf)$ -cycle. In contrast, G_{LB} is IMM-consistent.

2.2 An Informal Introduction to Weakestmo

We move on to *Weakestmo*, which also defines the program’s semantics as a set of execution graphs. However, they are constructed differently – extracted from a final *event structure*, which *Weakestmo* incrementally builds for a program.

¹ For a detailed formal description of the graphs and their construction process we refer the reader to [19, §2.2].



■ **Figure 3** A run of Weakestmo witnessing the annotated outcome of LB.

An event structure represents multiple executions of a program in a single graph. Like execution graphs, event structures contain a set of events and several relations among them. Like execution graphs, the *program order* (po) orders events according to each thread’s control flow. However, unlike execution graphs, po is not necessarily total among the events of a given thread. Events of the same thread that are not po -ordered are said to be in *conflict* (cf) with one another, and cannot belong to the same execution. Such conflict events arise when two read events originate from the same read instruction (e.g., representing executions where the reads return different values). Moreover, cf “extends downwards”: events that depend upon conflicting events (i.e., have conflicting po -predecessors) are also in conflict with one other. In pictures, we typically show only the *immediate conflict* edges (between reads originating from the same instruction) and omit the conflict edges between events po -after immediately conflicting ones.

Event structures are constructed incrementally starting from an event structure consisting only of the initialization events. Then, events corresponding to the execution of program instructions are added one at a time. We start by executing the first instruction of a program’s thread. Then, we may execute the second instruction of the same thread or the first instruction of another thread, and so on.

As an example, Fig. 3 constructs an event structure for LB. Fig. 3a depicts the event structure S_a obtained from the initial event structure by executing $a := [x]$ in LB’s thread 1. As a result of the instruction execution, a read event $e_{11}^1: R(x, 0)$ is added.

Whenever the event added is a read, Weakestmo has to justify the returned value from an appropriate write event. In this case, there is only one write to x – the initialization write – and so S_a has a *justified from* edge, denoted jf , going to e_{11}^1 in S_a . This is a requirement of Weakestmo: each read event in an event structure has to be justified from exactly one write

event with the same value and location. (This requirement is analogous to the *completeness* requirement in IMM-consistency for execution graphs.) Since events are added in program order and read events are always justified from existing events in the event structure, $\text{po} \cup \text{jf}$ is guaranteed to be acyclic by construction.

The next three steps (Figures 3b to 3d) simply add a new event to the event structure. Notice that unlike IMM executions, Weakestmo event structures do not track syntactic dependencies, e.g., S_d in Fig. 3d does not contain a **ppo** edge between e_1^2 and e_2^2 . This is precisely what allows Weakestmo to assign the same behavior to LB and LB-fake: they have exactly the same event structures. As a programming-language-level memory model, Weakestmo supports optimizations removing fake dependencies.

The next step (Fig. 3e) is more interesting because it showcases the key distinction between event structures and execution graphs, namely that event structures may contain more than one execution for each thread. Specifically, the transition from S_d to S_e reruns the first instruction of thread 1 and adds a new event e_{12}^1 justified from a different write event. We say that this new event *conflicts* (**cf**) with e_{11}^1 because they cannot both occur in a single execution. Because of conflicts, po in event structures does not totally order all events of a thread; e.g., e_{11}^1 and e_{12}^1 are not po -ordered in S_e . Two events of the same thread are conflicted precisely when they are not po -ordered.

The final construction step (Fig. 3f) demonstrates another Weakestmo feature. Conflicting write events writing the same value to the same location (e.g., e_{21}^1 and e_{22}^1 in S_f) may be declared *equal writes*, i.e., connected by an equivalence relation **ew**.²

The **ew** relation is used to define Weakestmo's version of the reads-from relation, **rf**, which relates a read to all (non-conflicted) writes *equal* to the write justifying the read. For example, e_1^2 reads from both e_{21}^1 and e_{22}^1 .

The Weakestmo's **rf** relation is used for extraction of program executions. An execution graph G is *extracted* from an event structure S denoted $S \triangleright G$ if G is a maximal conflict-free subset of S , it contains only *visible* events (to be defined in §3), and every read event in G reads from some write in G according to $S.\text{rf}$. Two execution graphs can be extracted from S_f : $\{\text{Init}, e_{11}^1, e_{21}^1, e_1^2, e_2^2\}$ and $\{\text{Init}, e_{12}^1, e_{22}^1, e_1^2, e_2^2\}$ representing the outcomes $a = 0 \wedge b = 1$ and $a = b = 1$ respectively.

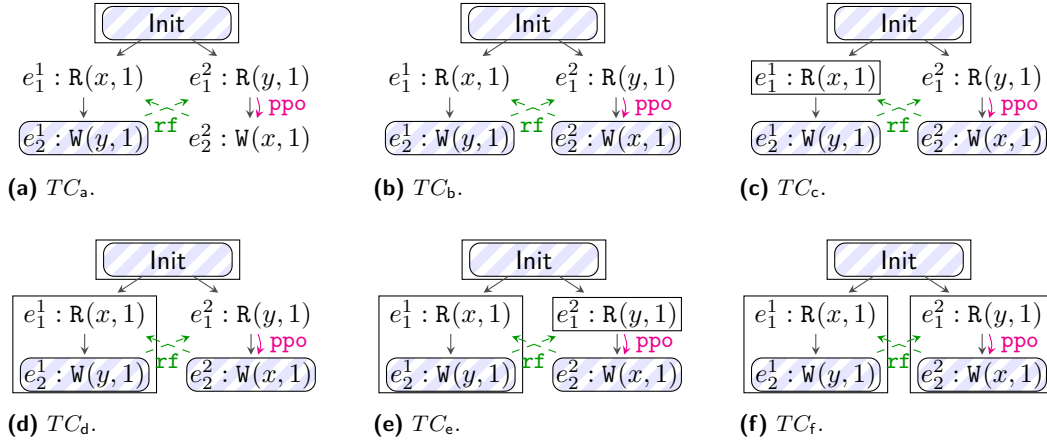
2.3 Weakestmo to IMM Compilation: High-Level Proof Structure

In this paper, we assume that Weakestmo is defined for the same assembly language as IMM (see [19, Fig. 2]) extended with SC accesses and refer to this language as L . Having that, we show the correctness of the *identity* mapping as a compilation scheme from Weakestmo to IMM in the following theorem.

► **Theorem 1.** *Let prog be a program in L , and G be an IMM-consistent execution graph of prog . Then there exists an event structure S of prog under Weakestmo such that $S \triangleright G$.*

To prove the theorem, we must show that Weakestmo may construct the needed event structure in a step by step fashion. If the IMM-consistent execution graph G contains no $\text{po} \cup \text{rf}$ cycles, then the construction is completely straightforward: G itself is a Weakestmo-consistent event structure (setting **jf** to be just **rf**), and its events can be added in any order extending $\text{po} \cup \text{rf}$.

² In this paper, we take **ew** to be reflexive, whereas it is irreflexive in Chakraborty and Vafeiadis [6]. Our **ew** is the reflexive closure of the one in [6].



■ **Figure 4** Traversal configurations for G_{LB} .

The construction becomes tricky for IMM-consistent execution graphs, such as G_{LB} , that contain $po \cup rf$ cycles. Due to the cycle(s), G cannot be directly constructed as a (conflict-free) Weakestmo event structure. We must instead construct a larger event structure S containing multiple executions, one of which will be the desired graph G . Roughly, for each $po \cup rf$ cycle in G , we have to construct an immediate conflict in the event structure.

To generate the event structure S , we rely on a basic property of IMM-consistent execution graphs shown by Podkopaev et al. [19, §§6,7], namely that execution graphs can be *traversed* in a certain order, i.e., its events can be *issued* and *covered* in that order, so that in the end all events are covered. The traversal captures a possible execution order of the program that yields the given execution. In that execution order, events are not added according to program order, but rather according to *preserved program order* (ppo) in two steps. Events are first issued when all their dependencies have been resolved, and are later covered when all their po-prior events have been covered.

In more detail, a traversal of an IMM-consistent execution graph G is a sequence of traversal steps between *traversal configurations*. A traversal configuration TC of an execution graph G is a pair of sets of events, $\langle C, I \rangle$, called the *covered* and *issued* set respectively. As an example, Fig. 4 presents all six traversal configurations of the execution graph G_{LB} of LB from Fig. 2a except for the initial configuration. The issued set is marked by \circ and the covered set by \square .

A traversal might be seen as an execution of an abstract machine that can execute write instructions early but has to execute everything else in order. The first option corresponds to issuing a write event, and the second option to covering an event. The traversal strategy has certain constraints. To issue a write event, all external reads that it depends upon must be resolved; i.e., they must read from already issued events. To cover an event, all its po-predecessors must also be covered.³ For example, in Fig. 4, a traversal cannot issue $e_2^2: W(x, 1)$ before issuing $e_2^1: W(y, 1)$ nor cover $e_1^1: R(x, 1)$ before issuing $e_2^2: W(x, 1)$.

According to Podkopaev et al. [19, Prop. 6.5], every IMM-consistent execution graph G has a full traversal of the following form:

$$G \vdash TC_{\text{init}}(G) \longrightarrow TC_1 \longrightarrow TC_2 \longrightarrow \dots \longrightarrow TC_{\text{final}}(G)$$

³ For readers familiar with PS [11], issuing a write event corresponds to promising a message, and covering an event to normal execution of an instruction.

where the initial configuration, $TC_{\text{init}}(G) \triangleq \langle G.\text{Init}, G.\text{Init} \rangle$, has issued and covered only G 's initial events and the final configuration, $TC_{\text{final}}(G) \triangleq \langle G.\text{E}, G.\text{W} \rangle$, has covered all G 's events and issued all its write events.

We construct the event structure S following a full traversal of G . We define a simulation relation, $\mathcal{I}(\text{prog}, G, TC, S, X)$, between the program prog , the current traversal configuration TC of execution G and the current event structure's state $\langle S, X \rangle$, where X is a subset of events corresponding to a particular execution graph extracted from the event structure S .

Our simulation proof is divided into the following three lemmas, which state that the initial states are simulated, that simulation extends along traversal steps, and that the simulation of final states means that G can be extracted from the generated event structure.

► **Lemma 2 (Simulation Start).** *Let prog be a program of \mathbb{L} , and G be an IMM-consistent execution graph of prog . Then $\mathcal{I}(\text{prog}, G, TC_{\text{init}}(G), S_{\text{init}}(\text{prog}), S_{\text{init}}(\text{prog}).\text{E})$ holds.*

► **Lemma 3 (Weak Simulation Step).** *If $\mathcal{I}(\text{prog}, G, TC, S, X)$ and $G \vdash TC \longrightarrow TC'$ hold, then there exist S' and X' such that $\mathcal{I}(\text{prog}, G, TC', S', X')$ and $S \rightarrow^* S'$ hold.*


► **Lemma 4 (Simulation End).** *If $\mathcal{I}(\text{prog}, G, TC_{\text{final}}(G), S, X)$ holds, then the execution graph associated with X is isomorphic to G .*

The proof of Theorem 1 then proceeds by induction on the length of the traversal $G \vdash TC_{\text{init}}(G) \rightarrow^* TC_{\text{final}}(G)$. Lemma 2 serves as the base case, Lemma 3 is the induction step simulating each traversal step with a number of event structure construction steps, and Lemma 4 concludes the proof.

The proofs of Lemmas 2 and 4 are technical but fairly straightforward. (We define \mathcal{I} in a way that makes these lemmas immediate.) In contrast, Lemma 3 is much more difficult to prove. As we will see, simulating a traversal step sometimes requires us to construct a new branch in the event structure, i.e., to add multiple events (see Section 4.3).

2.4 Weakestmo to IMM Compilation Correctness by Example

Before presenting any formal definitions, we conclude this overview section by showcasing the construction used in the proof of Lemma 3 on execution graph G_{LB} in Fig. 2a following the traversal of Fig. 4. We have actually already seen the sequence of event structures constructed in Fig. 3. Note that, even though Figures 3 and 4 have the same number of steps, there is no one-to-one correspondence between them as we explain below.

Consider the last event structure S_f from Fig. 3. A subset of its events X_f marked by , which we call a *simulated execution*, is a maximal conflict-free subset of S_f and all read events in X_f read from some write in X_f (i.e., are justified from a write deemed “equal” to some write in X_f). Then, by definition, X_f is extracted from S_f . Also, an execution graph induced by X_f is isomorphic to G_{LB} . That is, construction of S_f for LB shows that in Weakestmo it is possible to observe the same behavior as G_{LB} . Now, we explain how we construct S_f and choose X_f .

During the simulation, we maintain the relation $\mathcal{I}(\text{prog}, G, TC, S, X)$ connecting a program prog , its execution graph G , its traversal configuration TC , an event structure S , and a subset of its events X . Among other properties (presented in Section 4.2), the relation states that all issued and covered events of TC have exact counterparts in X , and that X can be extracted from S .

The initial event structure and X_{init} consist of only initial events. Then, following issuing of event $e_2^1: \text{W}(y, 1)$ in TC_a (see Fig. 4a), we need to add a branch to the event structure that has $\text{W}(y, 1)$ in it. Since Weakestmo requires adding events according to program order, we

first need to add a read event corresponding to “ $a := [x]$ ” of LB’s thread 1. Each read event in an event structure has to be justified from somewhere. In this case, the only write event to location x is the initial one. That is, the added read event e_{11}^1 is justified from it (see Fig. 3a). In the general case, having more than one option, we would choose a “safe” write event for an added read event to be justified from, i.e., the one which the corresponding branch is “aware” of already and being justified from which would not break consistency of the event structure. After that, a write event $e_{21}^1: W(y, 1)$ can be added po-after e_{11}^1 (see Fig. 3b), and $\mathcal{I}(\text{LB}, G_{\text{LB}}, TC_a, S_b, X_b)$ holds for $X_b = \{\text{Init}, e_{11}^1, e_{21}^1\}$.

Next, we need to simulate the second traversal step (see Fig. 4b), which issues $W(x, 1)$. As with the previous step, we first need to add a read event related to the first read instruction of LB’s thread 2 (see Fig. 3c). However, unlike the previous step, the added event e_1^2 has to get value 1, since there is a dependency between instructions in thread 2. As we mentioned earlier, the traversal strategy guarantees that $e_2^1: W(y, 1)$ is issued at the moment of issuing $e_2^2: W(x, 1)$, so there is the corresponding event in the event structure to justify the read event e_1^2 from. Now, the write event $e_2^2: W(y, 1)$ representing e_2^2 can be added to the event structure (see Fig. 3d) and $\mathcal{I}(\text{LB}, G_{\text{LB}}, TC_b, S_d, X_d)$ holds for $X_d = \{\text{Init}, e_{11}^1, e_{21}^1, e_1^2, e_2^2\}$.

In the third traversal step (see Fig. 4c), the read event $e_1^1: R(x, 1)$ is covered. To have a representative event for e_1^1 in the event structure, we add e_{12}^1 (see Fig. 3e). It is justified from e_2^2 , which writes the needed value 1. Also, e_{12}^1 represents an alternative to e_{11}^1 execution of the first instruction of thread 1, so the events are in conflict.

However, we cannot choose a simulated execution X related to TC_c and S_e by the simulation relation since X has to contain e_{12}^1 and a representative for $e_2^1: W(y, 1)$ (in S_e it is represented by e_{21}^1) while being conflict-free. Thus, the event structure has to make one other step (see Fig. 3f) and add the new event e_{22}^1 to represent $e_2^1: W(y, 1)$. Now, the simulated execution contains everything needed, $X_f = \{\text{Init}, e_{12}^1, e_{22}^1, e_1^2, e_2^2\}$.

Since X_f has to be extracted from S_f , every read event in X has to be connected via an **rf** edge to an event in X .⁴ To preserve the requirement, we connect the newly added event e_{22}^1 and e_{12}^1 via an **ew** edge, i.e., marking them to be equal writes.⁵ This induces an **rf** edge between e_{22}^1 and e_1^2 . That is, $\mathcal{I}(\text{LB}, G_{\text{LB}}, TC_c, S_f, X_f)$ holds.

To simulate the remaining traversal steps (Figures 4d to 4f), we do not need to modify S_f because it already contains counterparts for the newly covered events and, moreover, the execution graph associated with X_f is isomorphic to G_{LB} . That is, we just need to show that $\mathcal{I}(\text{LB}, G_{\text{LB}}, TC_d, S_f, X_f)$, $\mathcal{I}(\text{LB}, G_{\text{LB}}, TC_e, S_f, X_f)$, and $\mathcal{I}(\text{LB}, G_{\text{LB}}, TC_f, S_f, X_f)$ hold.

3 Formal Definition of Weakestmo

In this section, we introduce the notation used in the rest of the paper and define the Weakestmo memory model. For simplicity, we present only a minimal fragment of Weakestmo containing only relaxed reads and writes. For the definition of the full Weakestmo model, we refer the readers to Chakraborty and Vafeiadis [6] and to our Coq development [16].

Notation. Given relations R_1 and R_2 , we write $R_1 ; R_2$ for their sequential composition. Given relation R , we write $R^?$, R^+ and R^* to denote its reflexive, transitive and reflexive-transitive closures. We write **id** to denote the identity relation (i.e., $\text{id} \triangleq \{(x, x)\}$). For a set

⁴ Actually, it is easy to show that there could be only one such event since equal writes are in conflict and X is conflict-free.

⁵ Note that we could have left e_{22}^1 without any outgoing **ew** edges since the choice of equal writes for newly added events in Weakestmo is non-deterministic. However, that would not preserve the simulation relation.

A , we write $[A]$ to denote the identity relation restricted to A (that is, $[A] \triangleq \{\langle a, a \rangle \mid a \in A\}$). Hence, for instance, we may write $[A]; R; [B]$ instead of $R \cap (A \times B)$. We also write $[e]$ to denote $\{e\}$ if e is not a set.

Given a function $f: A \rightarrow B$, we denote by $=_f$ the set of f -equivalent elements: $(=_f \triangleq \{\langle a, b \rangle \in A \times A \mid f(a) = f(b)\})$. In addition, given a relation R , we denote by $R|_{=_f}$ the restriction of R to f -equivalent elements ($R|_{=_f} \triangleq R \cap =_f$), and by $R|_{\neq f}$ be the restriction of R to non- f -equivalent elements ($R|_{\neq f} \triangleq R \setminus =_f$).

3.1 Events, Threads and Labels

Events, $e \in \text{Event}$, and *thread identifiers*, $t \in \text{Tid}$, are represented by natural numbers. We treat the thread with identifier 0 as the *initialization* thread. We let $x \in \text{Loc}$ to range over *locations*, and $v \in \text{Val}$ over *values*.

A label, $l \in \text{Lab}$, takes one of the following forms:

- $\text{R}(x, v)$ – a read of value v from location x .
- $\text{W}(x, v)$ – a write of value v to location x .

Given a label l the functions typ , loc , val return (when applicable) its type (i.e., R or W), location and value correspondingly. When a specific function assigning labels to events is clear from the context, we abuse the notations R and W to denote the sets of all events labelled with the corresponding type. We also use subscripts to further restrict this set to a specific location (e.g., W_x denotes the set of write events operating on location x .)

3.2 Event Structures

An *event structure* S is a tuple $\langle \text{E}, \text{tid}, \text{lab}, \text{po}, \text{jf}, \text{ew}, \text{co} \rangle$ where:

- E is a set of events, i.e., $\text{E} \subseteq \text{Event}$.
- $\text{tid}: \text{E} \rightarrow \text{Tid}$ is a function assigning a thread identifier to every event. We treat events with the thread identifier equal to 0 as *initialization events* and denote them as Init , that is $\text{Init} \triangleq \{e \in \text{E} \mid \text{tid}(e) = 0\}$.
- $\text{lab}: \text{E} \rightarrow \text{Lab}$ is a function assigning a label to every event in E .
- $\text{po} \subseteq \text{E} \times \text{E}$ is a strict partial order on events, called *program order*, that tracks their precedence in the control flow of the program. Initialization events are po -before all other events, whereas non-initialization events can only be po -before events from the same thread.

Not all events of a thread are necessarily ordered by po . We call such po -unordered non-initialization events of the same thread *conflicting* events. The corresponding binary relation cf is defined as follows:

$$\text{cf} \triangleq ([\text{E} \setminus \text{Init}] ; =_{\text{tid}} ; [\text{E} \setminus \text{Init}]) \setminus (\text{po} \cup \text{po}^{-1})^?$$

- $\text{jf} \subseteq [\text{E} \cap \text{W}]; (=_{\text{loc}} \cap =_{\text{val}}); [\text{E} \cap \text{R}]$ is the *justified from* relation, which relates a write event to the reads it justifies. We require that reads are not justified by conflicting writes (i.e., $\text{jf} \cap \text{cf} = \emptyset$) and jf^{-1} be *functional* (i.e., whenever $\langle w_1, r \rangle, \langle w_2, r \rangle \in \text{jf}$, then $w_1 = w_2$). We also define the notion of *external* justification: $\text{jfe} \triangleq \text{jf} \setminus \text{po}$. A read event is externally justified from a write if the write is not po -before the read.
- $\text{ew} \subseteq [\text{E} \cap \text{W}]; (\text{cf} \cap =_{\text{loc}} \cap =_{\text{val}})^?; [\text{E} \cap \text{W}]$ is an equivalence relation called the *equal-writes* relation. Equal writes have the same location and value, and (unless identical) are in conflict with one another.

- $\text{co} \subseteq [\mathbf{E} \cap \mathbf{W}] ; (=_{\text{loc}} \setminus \text{ew}) ; [\mathbf{E} \cap \mathbf{W}]$ is the *coherence* order, a strict partial order that relates non-equal write events with the same location. We require that coherence be closed with respect to equal writes (i.e., $\text{ew} ; \text{co} ; \text{ew} \subseteq \text{co}$) and total with respect to ew on writes to the same location:

$$\forall x \in \text{Loc}. \forall w_1, w_2 \in \mathbf{W}_x. \langle w_1, w_2 \rangle \in \text{ew} \cup \text{co} \cup \text{co}^{-1}$$

Given an event structure S , we use “dot notation” to refer to its components (e.g., $S.\mathbf{E}$, $S.\text{po}$). For a set A of events, we write $S.A$ for the set $A \cap S.\mathbf{E}$ (for instance, $S.\mathbf{W}_x = \{e \in S.\mathbf{E} \mid \text{typ}(S.\text{lab}(e)) = \mathbf{W} \wedge \text{loc}(S.\text{lab}(e)) = x\}$). Further, for $e \in S.\mathbf{E}$, we write $S.\text{typ}(e)$ to retrieve $\text{typ}(S.\text{lab}(e))$. Similar notation is used for the functions loc and val . Given a set of thread identifiers T , we write $S.\text{thread}(T)$ to denote the set of events belonging to one of the threads in T , i.e., $S.\text{thread}(T) \triangleq \{e \in S.\mathbf{E} \mid S.\text{tid}(e) \in T\}$. When $T = \{\text{thread}(t)\}$ is a singleton, we often write $S.\text{thread}(t)$ instead of $S.\text{thread}(\{t\})$.

We define the immediate po and cf edges of an event structure as follows:

$$S.\text{po}_{\text{imm}} \triangleq S.\text{po} \setminus (S.\text{po} ; S.\text{po}) \quad S.\text{cf}_{\text{imm}} \triangleq S.\text{cf} \cap (S.\text{po}_{\text{imm}}^{-1} ; S.\text{po}_{\text{imm}})$$

An event e_1 is an immediate po -predecessor of e_2 if e_1 is po -before e_2 and there is no event po -between them. Two conflicting events are immediately conflicting if they have the same immediate po -predecessor.⁶

3.3 Event Structure Construction

Given a program prog , we construct its event structures operationally in a way that guarantees completeness (i.e., that every read is justified from some write) and $\text{po} \cup \text{jf}$ acyclicity. We start with an event structure containing only the initialization events and add one event at a time following each thread’s semantics.

For the thread semantics, we assume reductions of the form $\sigma \xrightarrow{e} \sigma'$ between thread states $\sigma, \sigma' \in \text{ThreadState}$ and labeled by the event $e \in \mathbf{E}$ generated by that execution step. Given a thread t and a sequence of events $e_1, \dots, e_n \in S.\text{thread}(t)$ in immediate po succession (i.e., $\langle e_i, e_{i+1} \rangle \in S.\text{po}_{\text{imm}}$ for $1 \leq i < n$) starting from a first event of thread t (i.e., $\text{dom}(S.\text{po}; [e_1]) \subseteq \text{Init}$), we can add an event e po -after that sequence of events provided that there exist thread states $\sigma_1, \dots, \sigma_n$ and σ' such that $\text{prog}(t) \xrightarrow{e_1} \sigma_1 \xrightarrow{e_2} \sigma_2 \dots \xrightarrow{e_n} \sigma_n \xrightarrow{e} \sigma'$, where $\text{prog}(t)$ is the initial thread state of thread t of the program prog . By construction, this means that the newly added event e will be in conflict with all other events of thread t besides e_1, \dots, e_n .

Further, when the new event e is a read event, it has to be justified from an existing write event, so as to ensure completeness and prevent “out-of-thin-air” values. The write event is picked non-deterministically from all non-conflicting writes with the same location as the new read event. Similarly, when e is a write event, its position in co order should be chosen. It can be done by either picking an ew equivalence class and including the new write in it, or by putting the new write immediately after some existing write in co order. At each step, we also check for *event structure consistency* (to be defined in Def. 5): If the event structure obtained after the addition of the new event is inconsistent, it is discarded.

⁶ Our definition of immediate conflicts differs from that of [6] and is easier to work with. The two definitions are equivalent if the set of initialization events is non-empty.

3.4 Event Structure Consistency

To define consistency, we first need a number of auxiliary definitions. The *happens-before* order $S.\mathbf{hb}$ is a generalization of the program order. Besides the program order edges, it includes certain *synchronization* edges (captured by the *synchronizes with* relation, $S.\mathbf{sw}$).

$$S.\mathbf{hb} \triangleq (S.\mathbf{po} \cup S.\mathbf{sw})^+$$

For the fragment covered in this section, there are no synchronization edges (i.e., $\mathbf{sw} = \emptyset$), and so \mathbf{hb} and \mathbf{po} coincide. In the full model,⁷ however, certain justification edges (e.g., between release/acquire accesses) contribute to \mathbf{sw} and hence to \mathbf{hb} .

The *extended conflict* relation $S.\mathbf{ecf}$ extends the notion of conflicting events to account for \mathbf{hb} ; two events are in extended conflict if they happen after conflicting events.

$$S.\mathbf{ecf} \triangleq (S.\mathbf{hb}^{-1})^? ; S.\mathbf{cf} ; S.\mathbf{hb}^?$$

As already mentioned in §2, the *reads-from* relation, $S.\mathbf{rf}$, of a Weakestmo event structure is derived. It is defined as an extension of $S.\mathbf{jf}$ to all $S.\mathbf{ew}$ -equivalent writes.

$$S.\mathbf{rf} \triangleq (S.\mathbf{ew} ; S.\mathbf{jf}) \setminus S.\mathbf{cf}$$

Note that unlike $S.\mathbf{jf}^{-1}$, the relation $S.\mathbf{rf}^{-1}$ is not functional. This does not cause any problems, however, since all the writes from whence a read reads have the same location and value and are in conflict with one another.

The relation $S.\mathbf{fr}$, called *from-read* or *reads-before*, places read events before subsequent writes.

$$S.\mathbf{fr} \triangleq S.\mathbf{rf}^{-1} ; S.\mathbf{co}$$

The *extended coherence* $S.\mathbf{eco}$ is a strict partial order that orders events operating on the same location. (It is almost total on accesses to a given location, except that it does not order equal writes nor reads reading from the same write.)

$$S.\mathbf{eco} \triangleq (S.\mathbf{co} \cup S.\mathbf{rf} \cup S.\mathbf{fr})^+$$

We observe that in our model, \mathbf{eco} is equal to $\mathbf{rf} \cup \mathbf{co} ; \mathbf{rf}^? \cup \mathbf{fr} ; \mathbf{rf}^?$, similar to the corresponding definitions about execution graphs in the literature.⁸

The last ingredient that we need for event structure consistency is the notion of *visible* events, which will be used to constrain external justifications. We define it in a few steps. Let e be some event in S . First, consider all write events used to externally justify e or one of its justification ancestors. The relation $S.\mathbf{jfe} ; (S.\mathbf{po} \cup S.\mathbf{jf})^*$ defines this connection formally. Among that set of write events restrict attention to those conflicting with e , and call that set M . That is, $M \triangleq \text{dom}(S.\mathbf{cf} \cap (S.\mathbf{jfe} ; (S.\mathbf{po} \cup S.\mathbf{jf})^*) ; [e])$. Event e is *visible* if all writes in M have an equal write that is \mathbf{po} -related with e . Formally,⁹

$$S.\mathbf{vis} \triangleq \{e \in S.E \mid S.\mathbf{cf} \cap (S.\mathbf{jfe} ; (S.\mathbf{po} \cup S.\mathbf{jf})^*) ; [e] \subseteq S.\mathbf{ew} ; (S.\mathbf{po} \cup S.\mathbf{po}^{-1})^?\}$$

Intuitively, visible events cannot depend on conflicting events: for every such justification dependence, there ought to be an equal non-conflicting write.

⁷ The full model is presented in [6] and also in our Coq development [16].

⁸ This equivalence equivalence does not hold in the original Weakestmo model [6]. To make the equivalence hold, we made \mathbf{ew} transitive, and require $\mathbf{ew} ; \mathbf{co} ; \mathbf{ew} \subseteq \mathbf{co}$.

⁹ Note, that in [6] the definition of the visible events is slightly more verbose. We proved in Coq [16] that our simpler definition is equivalent to the one given there.

Consistency places a number of additional constraints on event structures. First, it checks that there is no redundancy in the event structure: immediate conflicts arise only because of read events justified from non-equal writes. Second, it extends the constraints about **cf** to the extended conflict **ecf**; namely that no event can conflict with itself or be justified from a conflicting event. Third, it checks that reads are justified either from events of the same thread or from visible events of other threads. Finally, it ensures *coherence*, i.e., that executions restricted to accesses on a single location do not have any weak behaviors.

► **Definition 5.** An event structure S is said to be consistent if the following conditions hold.

- $\text{dom}(S.\text{cf}_{\text{imm}}) \subseteq S.\text{R}$ (**cf**_{imm}-READ)
- $S.\text{jf}; S.\text{cf}_{\text{imm}}; S.\text{jf}^{-1}; S.\text{ew}$ is irreflexive. (**cf**_{imm}-JUSTIFICATION)
- $S.\text{ecf}$ is irreflexive. (**ecf**-IRREFLEXIVITY)
- $S.\text{jf} \cap S.\text{ecf} = \emptyset$ (**jf**-NON-CONFLICT)
- $\text{dom}(S.\text{jfe}) \subseteq S.\text{Vis}$ (**jfe**-VISIBLE)
- $S.\text{hb}; S.\text{eco}^?$ is irreflexive. (COHERENCE)

3.5 Execution Extraction

The last part of *Weakestmo* is the extraction of executions from an event structure. An execution is essentially a conflict-free event structure.

► **Definition 6.** An execution graph G is a tuple $\langle E, \text{tid}, \text{lab}, \text{po}, \text{rf}, \text{co} \rangle$ where its components are defined similarly as in the case of an event structure with the following exceptions:

- po is required to be total on the set of events from the same thread. Thus, execution graphs have no conflicting events, i.e., $\text{cf} = \emptyset$.
- The **rf** relation is given explicitly instead of being derived. Also, there are no **jf** and **ew** relations.
- **co** totally orders write events operating on the same location.

All derived relations are defined similarly as for event structures. Next we show how to extract an execution graph from the event structure.

► **Definition 7.** A set of events X is called extracted from S if the following conditions are met:

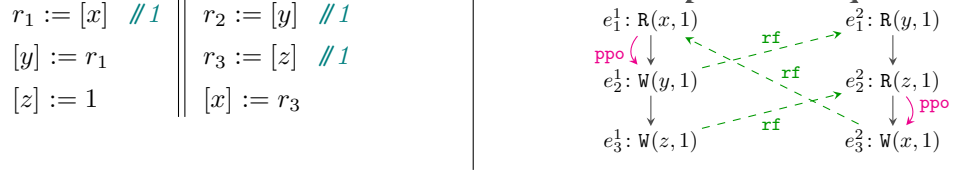
- X is conflict-free, i.e., $[X]; S.\text{cf}; [X] = \emptyset$.
- X is **S.rf**-complete, i.e., $X \cap S.\text{R} \subseteq \text{codom}([X]; S.\text{rf})$.
- X contains only visible events of S , i.e., $X \subseteq S.\text{Vis}$.
- X is **hb**-downward-closed, i.e., $\text{dom}(S.\text{hb}; [X]) \subseteq X$.

Given an event structure S and extracted subset of its events X , it is possible to associate with X an execution graph G simply by restricting the corresponding components of S to X :

$$\begin{array}{lll} G.E = X & G.\text{tid} = S.\text{tid}|_X & G.\text{lab} = S.\text{lab}|_X \\ G.\text{po} = [X]; S.\text{po}; [X] & G.\text{rf} = [X]; S.\text{rf}; [X] & G.\text{co} = [X]; S.\text{co}; [X] \end{array}$$

We say that such execution graph G is *associated with* X and that it is *extracted* from the event structure: $S \triangleright G$.

Weakestmo additionally defines another consistency predicate to further filter out some of the extracted execution graphs. In the *Weakestmo* fragment we consider, this additional consistency predicate is trivial – every extracted execution satisfies it – and so we do not present it here. In the full model, execution consistency checks atomicity of read-modify-write instructions, and sequential consistency for SC accesses.



■ **Figure 5** A variant of the load-buffering program (left) and the IMM graph G corresponding to its annotated weak behavior (right).

4 Compilation Proof for Weakestmo

In this section, we outline our correctness proof for the compilation from *Weakestmo* to the various hardware models. As already mentioned, our proof utilizes IMM [19]. In the following, we briefly present IMM for the fragment of the model containing only relaxed reads and writes (Section 4.1), our simulation relation (Section 4.2) for the compilation from *Weakestmo* to IMM, and outline the argument as to why the simulation relation is preserved (Section 4.3). Mapping from IMM to the hardware models has already been proved correct by Podkopaev et al. [19], so we do not present this part here. Later, in §5, we will extend the IMM mapping results to cover SC accesses.

As a further motivating example for this section consider yet another variant of the load buffering program shown in Fig. 5. As we will see, its annotated weak behavior is allowed by IMM and also by *Weakestmo*, albeit in a different way. The argument for constructing the *Weakestmo* event structure that exhibits the weak behavior from the given IMM execution graph is non-trivial.

4.1 The Intermediate Memory Model IMM

In order to discuss the proof, we briefly present a simplified version of the formal IMM definition, where we have omitted constraints about RMW accesses and fences.

► **Definition 8.** An IMM execution graph G is an execution graph (Def. 6) extended with one additional component: the preserved program order $\text{ppo} \subseteq [\text{R}] ; \text{po} ; [\text{W}]$.

Preserved program order edges correspond to syntactic dependencies guaranteed to be preserved by all major hardware platforms. For example, the execution graph in Fig. 5 has two *ppo* edges corresponding to the data dependencies via registers r_1 and r_3 . (The full IMM definition [19] distinguishes between the different types of dependencies – control, data, address–and includes them as separate components of execution graphs. In the full model, *ppo* is actually derived from the more basic dependencies.)

IMM-consistency checks completeness, coherence, and acyclicity:¹⁰

- **Definition 9.** An IMM execution graph G is IMM-consistent if
- $\text{codom}(G.\text{rf}) = G.\text{R}$, (COMPLETENESS)
 - $G.\text{hb} ; G.\text{eco}^2$ is irreflexive, and (COHERENCE)
 - $G.\text{rf} \cup G.\text{ppo}$ is acyclic. (NO-THIN-AIR)

¹⁰ Again, this is a simplified presentation for a fragment of the model. We refer the reader to Podkopaev et al. [19] for the full definition, which further distinguishes between internal and external *rf* edges.

As we can see, the execution graph G of Fig. 5 is IMM-consistent because every read of the graph reads from some write event and, moreover, the COHERENCE and NO-THIN-AIR properties hold.

4.2 Simulation Relation for Weakestmo to IMM Proof

In this section, we define the simulation relation \mathcal{I} ¹¹, which is used for the simulation of a traversal of an IMM-consistent execution graph by a Weakestmo event structure presented in Section 2.3.

The way we define $\mathcal{I}(prog, G, \langle C, I \rangle, S, X)$ induces a strong connection between events in the execution graph G and the event structure S . We make this connection explicit with the function $\mathbf{s2g}_{G,S} : S.E \rightarrow G.E$, which maps events of the event structure S into the events of the execution graph G , such that e and $\mathbf{s2g}_{G,S}(e)$ belong to the same thread and have the same po-position in the thread.¹² Note that $\mathbf{s2g}_{G,S}$ is defined for all events $e \in S.E$, meaning that the event structure S does not contain any redundant events that do not correspond to events in the IMM execution graph G . The function $\mathbf{s2g}_{G,S}$, however, does not have to be injective: in particular, events e and e' that are in immediate conflict in S have the same $\mathbf{s2g}_{G,S}$ -image in G . In the rest of the paper, whenever G and S are clear from the context, we omit the G, S subscript from $\mathbf{s2g}$.

In the context of a function $\mathbf{s2g}$ (for some G and S), we also use $\llbracket \cdot \rrbracket$ and $\llbracket \cdot \rrbracket$ to lift $\mathbf{s2g}$ to sets and relations:

$$\begin{aligned} \text{for } A_S \subseteq S.E : \llbracket A_S \rrbracket &\triangleq \{\mathbf{s2g}(e) \mid e \in A_S\} \\ \text{for } A_G \subseteq G.E : \llbracket A_G \rrbracket &\triangleq \{e \in S.E \mid \mathbf{s2g}(e) \in A_G\} \\ \text{for } R_S \subseteq S.E \times S.E : \llbracket R_S \rrbracket &\triangleq \{\langle \mathbf{s2g}(e), \mathbf{s2g}(e') \rangle \mid \langle e, e' \rangle \in R_S\} \\ \text{for } R_G \subseteq G.E \times G.E : \llbracket R_G \rrbracket &\triangleq \{\langle e, e' \rangle \in S.E \times S.E \mid \langle \mathbf{s2g}(e), \mathbf{s2g}(e') \rangle \in R_G\} \end{aligned}$$

For example, $\llbracket C \rrbracket$ denotes a subset of S 's events whose $\mathbf{s2g}$ -images are covered events in G , and $\llbracket S.rf \rrbracket$ denotes a relation on events in G whose $\mathbf{s2g}$ -preimages in S are related by $S.rf$.

We define the relation $\mathcal{I}(prog, G, \langle C, I \rangle, S, X)$ to hold if the following conditions are met:

1. G is an IMM-consistent execution of $prog$.
2. S is a Weakestmo-consistent event structure of $prog$.
3. X is an extracted subset of S .
4. S and X corresponds precisely to all covered and issued events and their po-predecessors:
 - $\llbracket S.E \rrbracket = \llbracket X \rrbracket = C \cup \text{dom}(G.po^? ; [I])$
 (Note that C is closed under po-predecessors, so $\text{dom}(G.po^? ; [C]) = C$.)

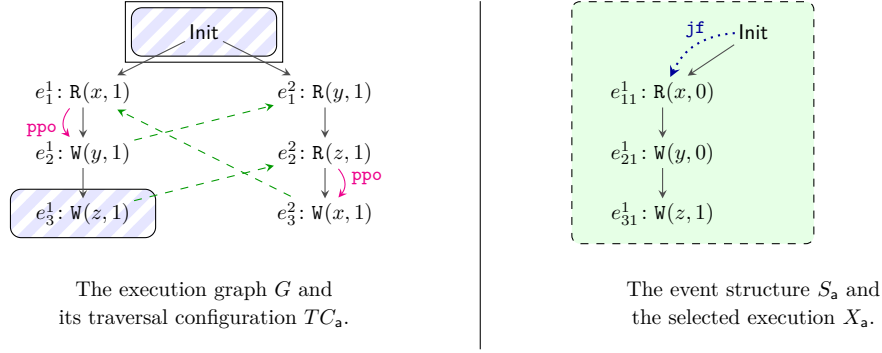
¹¹ A refined version of the simulation relation for the full Weakestmo model can be found in [16, Appendix A].

¹² Here we assume existence and uniqueness of such a function. In our Coq development [16], we have a different representation of execution graph events (but the same for events of event structures), which makes the existence and uniqueness questions trivial.

More specifically, we follow Podkopaev et al. [19, §2.2]. There each non-initializing event e of an execution graph G is encoded as a pair $\langle t, n \rangle$ where t is e 's thread and n is a serial number of e in thread t , i.e., a position of e in $G.po$ restricted to events of thread t ; each initializing event is encoded by the corresponding location - $\langle \text{init } l \rangle$.

In this representation, the function $\mathbf{s2g}_{G,S}$ for an event e returns (i) the e 's thread and a number of non-initial events which $S.po$ -preceded e if e is non-initializing or (ii) its location if it is initializing:

$$\mathbf{s2g}_{G,S}(e) \triangleq \begin{cases} \langle S.tid(e), |\text{dom}([S.E \setminus S.Init]; S.po; [e])| \rangle & \text{for } e \notin S.Init \\ \langle \text{init } S.loc(e) \rangle & \text{for } e \in S.Init \end{cases}$$

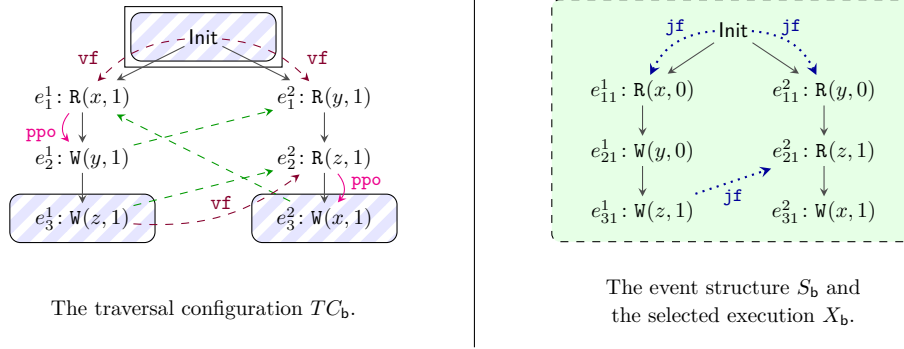


■ **Figure 6** The execution graph G , its traversal configuration TC_a , the related event structure S_a , and the selected execution X_a . Covered events are marked by \square and issued ones by \circ . Events belonging to the selected execution are marked by \odot .

5. Each S event has the same thread, type, modifier, and location as its corresponding G event. In addition, covered and issued events in X have the same value as their corresponding ones in G .
 - a. $\forall e \in S.E. S.\{\mathbf{tid}, \mathbf{typ}, \mathbf{loc}, \mathbf{mod}\}(e) = G.\{\mathbf{tid}, \mathbf{typ}, \mathbf{loc}, \mathbf{mod}\}(s2g(e))$
 - b. $\forall e \in X \cap \llbracket C \cup I \rrbracket. S.\mathbf{val}(e) = G.\mathbf{val}(s2g(e))$
6. Program order in S corresponds to program order in G :
 - $\llbracket S.\mathbf{po} \rrbracket \subseteq G.\mathbf{po}$
7. Identity relation in G corresponds to identity or conflict relation in S :
 - $\llbracket \mathbf{id} \rrbracket \subseteq S.\mathbf{cf}?$
8. Reads in S are justified by writes that have already been observed by the corresponding events in G . Moreover, covered events in X are justified by a write corresponding to that read from the corresponding read in G :
 - a. $\llbracket S.\mathbf{jf} \rrbracket \subseteq G.\mathbf{rf}?$; $G.\mathbf{hb}?$
 - b. $\llbracket S.\mathbf{jf}; [X \cap \llbracket C \rrbracket] \rrbracket \subseteq G.\mathbf{rf}$
9. Every write event justifying some external read event should be $S.\mathbf{ew}$ -equal to some issued write event in X :
 - $dom(S.\mathbf{jfe}) \subseteq dom(S.\mathbf{ew}; [X \cap \llbracket I \rrbracket])$
10. Equal writes in S correspond to the same write event in G :
 - $\llbracket S.\mathbf{ew} \rrbracket \subseteq \mathbf{id}$
11. Every non-trivial $S.\mathbf{ew}$ equivalence class contains an issued write in X :
 - $S.\mathbf{ew} \subseteq (S.\mathbf{ew}; [X \cap \llbracket I \rrbracket]; S.\mathbf{ew})?$
12. Coherence edges in S correspond to coherence or identity edges in G . (We will explain in Section 4.3 why a coherence edge in S might correspond to an identity edge in G .)
 - $\llbracket S.\mathbf{co} \rrbracket \subseteq G.\mathbf{co}?$

As an example, consider the execution G from Fig. 5, the traversal configuration $TC_a \triangleq \{\{\mathbf{Init}\}, \{\mathbf{Init}, e_3^1\}\}$, and the event structure S_a shown in Fig. 6. We will show that $\mathcal{I}(prog, G, TC_a, S_a, X_a)$, where $X_a \triangleq S_a.E$, holds.

Take $s2g_{G, S_a} = \{\mathbf{Init} \mapsto \mathbf{Init}, e_{11}^1 \mapsto e_1^1, e_{21}^1 \mapsto e_2^1, e_{31}^1 \mapsto e_3^1\}$. Given that $\mathbf{cf} = \mathbf{ew} = \emptyset$, the consistency constraints hold immediately. For example, condition 8 holds because e_{11}^1 is justified by \mathbf{Init} , which happens before it. Finally, note that only e_{31}^1 and e_3^1 are required to have the same value by constraint 5, the other related thread events only need to have the same type and address.



■ **Figure 7** The traversal configuration TC_b , the related event structure S_b , and the selected execution X_b .

The definition of the simulation relation \mathcal{I} renders the proofs of Lemmas 2 and 4 straightforward. Specifically, for Lemma 2, the initial configuration $TC_{\text{init}}(G)$ containing only the initialization events is simulated by the initial event structure S_{init} as all the constraints are trivially satisfied ($S_{\text{init}}.\text{po} = S_{\text{init}}.\text{jf} = S_{\text{init}}.\text{ew} = S_{\text{init}}.\text{co} = \emptyset$).

For Lemma 4, since $TC_{\text{final}}(G)$ covers all events of G , property 5 implies that the labels of the events in X are equal to the corresponding events of G ; property 6 means that po is the same between them; property 8 means that rf is the same between them; properties 7 and 12 together mean that co is the same. Therefore, G and the execution corresponding to X are isomorphic.

4.3 Simulation Step Proof Outline

We next outline the proof of Lemma 3, which states that the simulation relation \mathcal{I} can be restored after a traversal step.

Suppose that $\mathcal{I}(\text{prog}, G, TC, S, X)$ holds for some prog , G , TC , S , and X , and we need to simulate a traversal step $TC \rightarrow TC'$ that either covers or issues an event of thread t . Then we need to produce an event structure S' and a subset of its events X' such that $\mathcal{I}(\text{prog}, G, TC', S', X')$ holds. Whenever thread t has any uncovered issued write events, Weakestmo might need to take multiple steps from S to S' so as to add any missing events po -before the uncovered issued writes of thread t . Borrowing the terminology of the “promising semantics” [11], we refer to these steps as constructing a certification branch for the issued write(s).

Before we present the construction, let us return to the example of Fig. 5. Consider the traversal step from configuration TC_a to configuration $TC_b \triangleq \{\{\text{Init}\}, \{\text{Init}, e_3^1, e_3^2\}\}$ by issuing the event e_3^2 (see Fig. 7). To simulate this step, we need to show that it is possible to execute instructions of thread 2 and extend the event structure with a set of events Br_b matching these instructions. As we have already seen, the labels of the new events can differ from their counterparts in G – they only have to agree for the covered and issued events. In this case, we set $Br_b = \{e_{11}^2, e_{21}^2, e_{31}^2\}$, and adding them to the event structure S_a gives us event structure S_b shown in Fig. 7.

In more detail, we need to build a run of thread-local semantics $\text{prog}(2) \xrightarrow{e_{11}^2} \xrightarrow{e_{21}^2} \xrightarrow{e_{31}^2} \sigma'$ such that (1) it contains events corresponding to all the events of thread 2 up to e_3^2 (i.e., e_1^2, e_2^2, e_3^2) with the same location, type, and thread identifier and (2) any events corresponding to covered or issued events (i.e., e_3^2) should also have the same value as the corresponding event in G .

Then, following the run of the thread-local semantics, we should extend the event structure S_a to S_b by adding new events Br_b , and ensure that the constructed event structure S_b is consistent (Def. 5) and simulates the configuration TC_b . In particular, it means that:

- for each read event in Br_b we need to pick a justification write event, which is either already present in S or po -preceed the read event;
- for each write event in Br_b we should determine its position in co order of the event structure.

Finally, we need to update the selected execution by replacing all events of thread 2 by the new events Br_b : $X_b \triangleq X_a \setminus S.\text{thread}(\{2\}) \cup Br_b$.

4.3.1 Justifying the New Read Events

In order to determine whence these read events should be justified (and hence what value they should return), we have adopted the approach of Podkopaev et al. [19] for a similar problem with certifying promises in the compilation proof from PS to IMM. The construction relies on several auxiliary definitions.

First, given an execution G and a traversal configuration $\langle C, I \rangle$, we define the set of *determined* events to be those events of G that must have equal counterparts in S . In particular, this means that S should assign to these events the same label as G , and thus the same reads-from source for the read events.

$$G.\text{determined}_{\langle C, I \rangle} \triangleq C \cup I \cup \text{dom}((G.\text{rf} \cap G.\text{po})^? ; G.\text{ppo} ; [I]) \cup \text{codom}([I] ; (G.\text{rf} \cap G.\text{po}))$$

Besides covered and issued events, the set of determined events also contains the ppo -prefixes of issued events, since issued events may depend on their values, as well as any internal reads reading from issued events, since their values are also determined by the issued events.

For the graph G and traversal configuration TC_b , the set of determined events contains events e_3^1 , e_2^2 , and e_3^2 . (The events e_3^1 and e_3^2 are issued, whereas e_2^2 has a ppo edge to e_3^2 .) In contrast, events e_1^1 , e_2^1 , and e_1^2 are not determined, since their corresponding events in S read/write a different value.

Second, we introduce the *viewfront* relation (vf) to contain all the writes that have been observed at a certain point in the graph. That is, the edge $\langle w, e \rangle \in G.\text{vf}_{TC}$ indicates that the write w either happens before e , is read by a covered event happening before e , or is read by a determined read earlier in the same thread as e .

$$G.\text{vf}_{\langle C, I \rangle} \triangleq [G.\text{w}] ; (G.\text{rf} ; [C])^? ; G.\text{hb}^? \cup G.\text{rf} ; [G.\text{determined}_{\langle C, I \rangle}] ; G.\text{po}^?$$

Figure 7 depicts three $G.\text{vf}_{TC_b}$ edges. Since $G.\text{vf}_{TC} ; G.\text{po} \subseteq G.\text{vf}_{TC}$, the other incoming viewfront edges to thread 2 can be derived. Note that there is no edge from e_2^1 to thread 2, since e_2^1 neither happens before any event in thread 2 nor is read by any determined read.

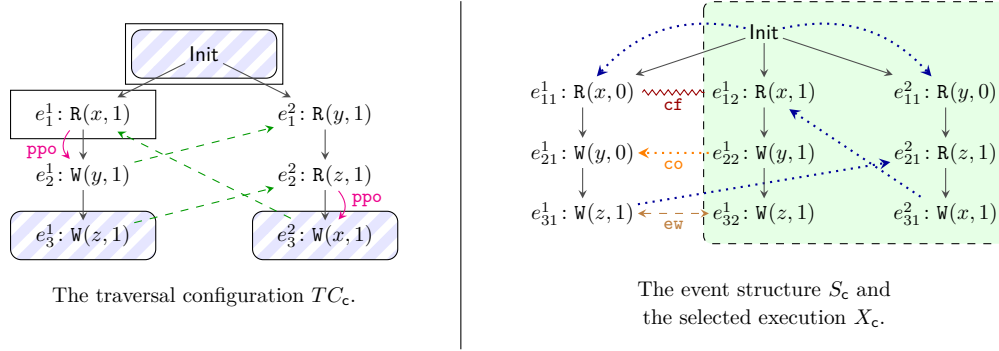
Finally, we construct the *stable justification* relation (sjf) that helps us justify the read events in Br_b in the event structure:

$$G.\text{sjf}_{TC} \triangleq ([G.\text{w}] ; (G.\text{vf}_{TC} \cap =_{G.\text{loc}}) ; [G.\text{r}]) \setminus (G.\text{co} ; G.\text{vf}_{TC})$$

It relates a read event r to the co -last “observed” write event with same location. Assuming that G is IMM-consistent, it can be shown that $G.\text{sjf}$ agrees with $G.\text{rf}$ on the set of determined reads.

$$G.\text{sjf}_{TC} ; [G.\text{determined}_{TC}] \subseteq G.\text{rf}$$

For the graph G and traversal configuration TC_b shown in Fig. 7 the sjf relation coincides with the depicted vf edges: i.e., we have $\langle \text{Init}, e_1^1 \rangle, \langle \text{Init}, e_1^2 \rangle, \langle e_3^1, e_2^2 \rangle \in G.\text{sjf}_{TC_b}$.



■ **Figure 8** The traversal configuration TC_c , the related event structure S_c , and the selected execution X_c .

Having \mathbf{sjf}_{TC_b} as a guide for values read by instructions in the certification run, we construct the steps of the thread-local operational semantics $\mathit{prog}(2) \rightarrow^* \sigma'$ using the receptiveness property of the thread's semantics, which essentially says that given an execution trace $\tau = e_1, \dots, e_n$ of the thread semantics, and a subset of events $K \subseteq \{e_1, \dots, e_{n-1}\}$ along that trace that have no **pp0**-successors in the graph, we arbitrarily change the values of read events in K , and there exist values for the write events in K such that the updated execution trace is also a trace of the thread semantics.¹³

The relation \mathbf{sjf}_{TC_b} is also used to pick justification writes for the read events in Br_b . We have proved that each **sjf** edge either starts in some issued event (of the previous traversal configuration) or it connects two events that are related by **po**:

$$G.\mathbf{sjf}_{TC_b} \subseteq [I_a]; G.\mathbf{sjf}_{TC_b} \cup G.\mathbf{po}$$

In the former case, thanks to the property 4 of our simulation relation, we can pick a write event from X_a corresponding to the issued write (e.g., for Fig. 7, it is the event e_{31}^1 , corresponding to the issued write e_3^1). In the latter case, we pick either the initial write or some $S_b.\mathbf{po}$ preceding write belonging to Br_b .

4.3.2 Ordering the New Write Events

In order to pick the $S_b.\mathbf{co}$ position of the new write events in the updated event structure, we generally follow the original $G.\mathbf{co}$ order of the IMM graph. Because of the conflicting events, however, it is not always possible to preserve the inclusion between the relations. This is why we relax the inclusion to $\llbracket S.\mathbf{co} \rrbracket \subseteq G.\mathbf{co}$? in property 12 of the simulation relation.

To see the problem let us return to the example. Suppose that the next traversal step covers the read e_1^1 . To simulate this step, we build an event structure S_c (see Fig. 8). It contains the new events $Br_c \triangleq \{e_{12}^1, e_{22}^1, e_{32}^1\}$.

Consider the write events e_{21}^1 and e_{22}^1 of the event structure. Since the events have different labels, we cannot make them **ew**-equivalent. And since $S_c.\mathbf{co}$ should be total among all writes to the same location (with respect to $S_c.\mathbf{ew}$), we must put a **co** edge between these two events in one direction or another. Note that events e_{21}^1 and e_{22}^1 correspond to the same event e_2^1 in the graph, thus we cannot use the coherence order of the graph $G.\mathbf{co}$ to guide our decision.

¹³The formal definition of the receptiveness property is quite elaborate. For the detailed definition we refer the reader to the Coq development of IMM [7].

In fact, the `co`-order between these two events does not matter, so we could pick either direction. For the purposes of our proofs, however, we found it more convenient to always put the new events earlier in the `co` order (thus we have $\langle e_{22}^1, e_{21}^1 \rangle \in S_c.\text{co}$). Thereby we can show that the `co` edges of the event structure ending in the new events, have corresponding edges in the graph: $\llbracket S_c.\text{co}; [Br_c] \rrbracket \subseteq G.\text{co}$.

Now consider the events e_{31}^1 and e_{32}^1 . Since these events have the same label and correspond to the same event in G , we make them `ew`-equivalent. In fact, this choice is necessary for the correctness of our construction. Otherwise, the new events Br_c would be deemed invisible, because of the $S_c.\text{cf} \cap (S_c.\text{jfe}; (S_c.\text{po} \cup S_c.\text{jf})^*)$ path between e_{31}^1 and e_{12}^1 . Recall that only the visible events can be used to extract an execution from the event structure (Def. 7).

In general, assuming that $\mathcal{I}(\text{prog}, G, \langle C, I \rangle, S, X)$ holds, we attach the new write event e to an $S.\text{ew}$ equivalence class represented by the write event w , s.t. (i) w has the same `s2g` image as e , i.e., $\text{s2g}(w) = \text{s2g}(e)$; (ii) w belongs to X and its `s2g` image is issued, that is $w \in X \cap \llbracket I \rrbracket$. If there is no such an event w , we put e `S.co`-after events such that their `s2g` images are ordered $G.\text{co}$ -before $\text{s2g}(e)$, and `S.co`-before events such that their `s2g` images are equal to $\text{s2g}(e)$ or ordered $G.\text{co}$ -after it. Note that thanks to property 9 of the simulation relation, that is $\text{dom}(S.\text{jfe}) \subseteq \text{dom}(S.\text{ew}; [X \cap \llbracket I \rrbracket])$, our choice of `ew` guarantees that all new events will be visible.

4.3.3 Construction Overview

To sum up, to prove Lemma 3, we consider the events of $G.\text{thread}(\{t\})$ where t is the thread of the event issued or covered by the traversal step $TC \rightarrow TC'$, together with the `sjf` relation determining the values of the read events. At this point, we can show that \mathcal{I} -conditions for the new configuration TC' hold for all events except for those in thread t .

Because of receptiveness, there exists a sequence of the thread steps $\text{prog}(t) \rightarrow^* \sigma'$ for some thread state σ' such that the labels on this sequence match the events $G.\text{thread}(\{t\})$ with the labels determined by `sjf`, and include an event with the same label as the one issued or covered by the traversal step $TC \rightarrow TC'$.

We then do an induction on this sequence of steps, and add each event to the event structure S and to its selected subset of events X (unless already there), showing along the way that the \mathcal{I} -conditions also hold for the updated event structure, selected subset, and the events added. At the end, when we have considered all the events generated by the step sequence, we will have generated the event structure S' and execution X' such that $\mathcal{I}(\text{prog}, G, TC', S', X')$ holds.

5 Handling SC Accesses

In this section, we briefly describe the changes needed in order to handle the compilation of Weakestmo's sequentially consistent (SC) accesses. The purpose of SC accesses is to guarantee sequential consistency for the simple programming pattern that uses exclusively SC accesses to communicate between threads. As Lahav et al. [13] showed, however, their semantics is quite complicated because they can be freely mixed with non-SC accesses.

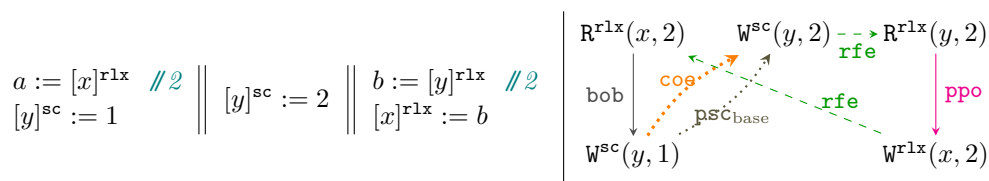
We first define an extension of IMM, which we call IMM_{SC} . Its consistency extends that of IMM with an additional acyclicity requirement concerning SC accesses, which is taken directly from RC11-consistency [13, Definition 1].

► **Definition 10.** An execution graph G is IMM_{SC} -consistent if it is IMM-consistent [19, Definition 3.11] and $G.\text{psc}_{\text{base}} \cup G.\text{psc}_{\text{F}}$ is acyclic, where:¹⁴

$$\begin{aligned} G.\text{scb} &\triangleq G.\text{po} \cup G.\text{po}|_{\neq G.\text{loc}}; G.\text{hb}; G.\text{po}|_{\neq G.\text{loc}} \cup G.\text{hb}|_{=\text{loc}} \cup G.\text{co} \cup G.\text{fr} \\ G.\text{psc}_{\text{base}} &\triangleq ([G.\text{E}^{\text{sc}}] \cup [G.\text{F}^{\text{sc}}]; G.\text{hb}^?); G.\text{scb}; ([G.\text{E}^{\text{sc}}] \cup G.\text{hb}^?); [G.\text{F}^{\text{sc}}] \\ G.\text{psc}_{\text{F}} &\triangleq [G.\text{F}^{\text{sc}}]; (G.\text{hb} \cup G.\text{hb}; G.\text{eco}; G.\text{hb}); [G.\text{F}^{\text{sc}}] \end{aligned}$$

The scb , psc_{base} and psc_{F} relations were carefully designed by Lahav et al. [13] (and recently adopted by the C++ standard), so that they provide strong enough guarantees for programmers while being weak enough to support the intended compilation of SC accesses to commodity hardware. In particular, a previous (simpler) proposal in [2], which essentially includes $G.\text{hb}$ between SC accesses in the relation required to be acyclic, is too strong for efficient compilation to the POWER architecture. Indeed, the compilation schemes to POWER do not enforce a strong barrier on hb -paths between SC accesses, but rather on $G.\text{po}; G.\text{hb}; G.\text{po}$ -paths between SC accesses.

► **Remark 11.** The full IMM model (i.e., including release/acquire accesses and SC fences, as defined by Podkopaev et al. [19]) forbids cycles in $\text{rfe} \cup \text{ppo} \cup \text{bob} \cup \text{psc}_{\text{F}}$, where bob is (similar to ppo) a subset of the program order that must be preserved due to the presence of a memory fence or release/acquire access. Since psc_{F} is already included in IMM’s acyclicity constraint, one may consider the natural option of including psc_{base} in that acyclicity constraint as well. However, it leads to a model that is too strong, as it forbids the following behavior:



This behavior is allowed by POWER (using any of the two intended compilation schemes for SC accesses; see Section 5.1.2).

Adapting the compilation from Weakestmo to IMM_{SC} to cover SC accesses is straightforward because the full definition of Weakestmo [6] does not have any additional constraints about SC accesses at the level of event structures. It only has an SC constraint at the level of extracted executions which is actually the same as in RC11, which we took as is for IMM_{SC} .

5.1 Compiling IMM_{SC} to Hardware

In this section, we establish describe the extension of the results of [19] to support SC accesses with their intended compilation schemes to the different architectures.

As was done in [19], since IMM_{SC} and the models of hardware we consider are all defined in the same declarative framework (using execution graphs), we formulate our results on the level of execution graphs. Thus, we actually consider the mapping of IMM_{SC} execution graphs to target architecture execution graphs that is induced by compilation of IMM_{SC} programs to machine programs. Hence, roughly speaking, for each architecture $\alpha \in \{\text{TSO}, \text{POWER}, \text{ARMv7}, \text{ARMv8}\}$, our (mechanized) result takes the following form:

¹⁴In IMM_{SC} , event labels include an “access mode”, where sc denotes an SC access. The sets $G.\text{E}^{\text{sc}}$ consists of all SC accesses (reads, writes and fences) in G , and $G.\text{F}^{\text{sc}}$ consists of all SC fences in G .

If the α -execution-graph G_α corresponds to the IMM_{SC} -execution-graph G , then α -consistency of G_α implies IMM_{SC} -consistency of G .

Since the mapping from **Weakestmo** to IMM_{SC} (on the program level) is the *identity mapping* (Theorem 1), we obtain as a corollary the correctness of the compilation from **Weakestmo** to each architecture α that we consider. The exact notions of correspondence between G_α and G are presented in [16, Appendicies B, C and D].

The mapping of IMM_{SC} to each architecture follows the intended compilation scheme of C/C++11 [15, 13], and extends the corresponding mappings of **IMM** from Podkopaev et al. [19] with the mapping of SC reads and writes. Next, we schematically present these extensions.

5.1.1 TSO

There are two alternative sound mappings of SC accesses to x86-TSO:

Fence after SC writes	Fence before SC reads
$(\text{R}^{\text{sc}}) \triangleq \text{mov}$	$(\text{R}^{\text{sc}}) \triangleq \text{mfence}; \text{mov}$
$(\text{W}^{\text{sc}}) \triangleq \text{mov}; \text{mfence}$	$(\text{W}^{\text{sc}}) \triangleq \text{mov}$
$(\text{RMW}^{\text{sc}}) \triangleq (\text{lock}) \text{ xchg}$	$(\text{RMW}^{\text{sc}}) \triangleq (\text{lock}) \text{ xchg}$

The first, which is implemented in mainstream compilers, inserts an **mfence** after every SC write; whereas the second inserts an **mfence** before every SC read. Importantly, one should *globally* apply one of the two mappings to ensure the existence of an **mfence** between every SC write and following SC read.

5.1.2 POWER

There are two alternative sound mappings of SC accesses to POWER:

Leading sync	Trailing sync
$(\text{R}^{\text{sc}}) \triangleq \text{sync}; (\text{R}^{\text{acq}})$	$(\text{R}^{\text{sc}}) \triangleq \text{ld}; \text{sync}$
$(\text{W}^{\text{sc}}) \triangleq \text{sync}; \text{st}$	$(\text{W}^{\text{sc}}) \triangleq (\text{W}^{\text{rel}}); \text{sync}$
$(\text{RMW}^{\text{sc}}) \triangleq \text{sync}; (\text{RMW}^{\text{acq}})$	$(\text{RMW}^{\text{sc}}) \triangleq (\text{RMW}^{\text{rel}}); \text{sync}$

The first scheme inserts a **sync** before every SC access, while the second inserts an **sync** after every SC access. Importantly, one should *globally* apply one of the two mappings to ensure the existence of a **sync** between every two SC accesses.

Observing that **sync** is the result of mapping an SC-fence to POWER, we can reuse the existing proof for the mapping of **IMM** to POWER. To handle the leading **sync** (respectively, trailing **sync**) scheme we introduce a preceding step, in which we prove that splitting in the whole execution graph each SC access to a pair of an SC fence followed (preceded) by a release/acquire access is a sound transformation under IMM_{SC} . That is, this global execution graph transformation cannot make an inconsistent execution consistent:

► **Theorem 12.** *Let G be an execution graph such that*

$$[\text{R}^{\text{sc}} \cup \text{W}^{\text{sc}}]; (G.\text{po}' \cup G.\text{po}'; G.\text{hb}; G.\text{po}'); [\text{R}^{\text{sc}} \cup \text{W}^{\text{sc}}] \subseteq G.\text{hb}; [\text{F}^{\text{sc}}]; G.\text{hb},$$

where $G.\text{po}' \triangleq G.\text{po} \setminus G.\text{rmw}$. Let G' be the execution graph obtained from G by weakening the access modes of SC write and read events to release and acquire modes respectively. Then, IMM_{SC} -consistency of G follows from **IMM**-consistency of G' .

Having this theorem, we can think about mapping of IMM_{SC} to POWER as if it consists of three steps. We establish the correctness of each of them separately.

1. At the IMM_{SC} level, we globally split each SC-access to an SC-fence and release/acquire access. Correctness of this step follows by Theorem 12.
2. We map IMM to POWER, whose correctness follows by the existing results of [19], since we do not have SC accesses at this stage.
3. We remove any redundant fences introduced by the previous step. Indeed, following the leading `sync` scheme, we will obtain `sync;lwsync;st` for an SC write. The `lwsync` is redundant here since `sync` provides stronger guarantees than `lwsync` and can be removed. Similarly, following the trailing `sync` scheme, we will obtain `ld;cmp;bc;isync;sync` for an SC read. Again, the `sync` makes other synchronization instructions redundant.

5.1.3 ARMv7

The ARMv7 model [1] is very similar to the POWER model with the main difference being that it has a weaker preserved program order than POWER. However, Podkopaev et al. [19] proved IMM to POWER compilation correctness without relying on POWER's preserved program order explicitly, but assuming the weaker version of ARMv7's order. Thus, their proof also establishes correctness of compilation from IMM to ARMv7.

Extending the proof to cover SC accesses follows the same scheme discussed for POWER, since two intended mappings of SC accesses for ARMv7 are the same except for replacing POWER's `sync` fence with ARMv7's `dmb`:

Leading <code>dmb</code>	Trailing <code>dmb</code>
$(\text{R}^{\text{sc}}) \triangleq \text{dmb}; (\text{R}^{\text{acq}})$	$(\text{R}^{\text{sc}}) \triangleq \text{ldr}; \text{dmb}$
$(\text{W}^{\text{sc}}) \triangleq \text{dmb}; \text{str}$	$(\text{W}^{\text{sc}}) \triangleq (\text{W}^{\text{rel}}); \text{dmb}$
$(\text{RMW}^{\text{sc}}) \triangleq \text{dmb}; (\text{RMW}^{\text{acq}})$	$(\text{RMW}^{\text{sc}}) \triangleq (\text{RMW}^{\text{rel}}); \text{dmb}$

5.1.4 ARMv8

Since ARMv8 has added dedicated instructions to support C/C++-style SC accesses, we have established the correctness of a mapping employing these new instructions:

(R^{sc})	$\triangleq \text{LDAR}$
(W^{sc})	$\triangleq \text{STLR}$
$(\text{FADD}^{\text{sc}})$	$\triangleq \text{L}; \text{LDAXR}; \text{STLXR}; \text{BC L}$
(CAS^{sc})	$\triangleq \text{L}; \text{LDAXR}; \text{CMP}; \text{BC Le}; \text{STLXR}; \text{BC L}; \text{Le}$

We note that in this mapping, we follow Podkopaev et al. [19] and compile RMW operations to loops with load-linked and store-conditional instructions (LDX/STX). An alternative mapping for RMWs would be to use single hardware instructions, such as LDADD and CAS, that directly implement the required functionality. Unfortunately, however, due to a limitation of the current IMM setup and unclarity about the exact semantics of the CAS instruction, we are not able to prove the correctness of the alternative mapping employing these instructions. The problem is that IMM assumes that every po-edge from a RMW instruction is preserved, which holds for the mapping of CAS using the aforementioned loop, but not necessarily using the single instruction.

6 Related Work

While there are several memory model definitions both for hardware architectures [1, 9, 17, 21, 22] and programming languages [3, 4, 10, 14, 18, 20] in the literature, there are relatively few compilation correctness results [6, 8, 11, 13, 19, 23].

Most of these compilation results do not tackle any of the problems caused by $\text{po} \cup \text{rf}$ cycles, which are the main cause of complexity in establishing correctness of compilation mappings to hardware architectures. A number of papers (e.g., [6, 11, 23]) consider only hardware models that forbid such cycles, such as x86-TSO [17] and “strong POWER” [12], while others (e.g., [8]) consider compilation schemes that introduce fences and/or dependencies so as to prevent $\text{po} \cup \text{rf}$ cycles. The only compilation results where there is some non-trivial interplay of dependencies are by Lahav et al. [13] and by Podkopaev et al. [19].

The former paper [13] defines the RC11 model (repaired C11), and establishes a number of results about it, most of which are not related to compilation. The only relevant result is its pencil-and-paper correctness proof of a compilation scheme from RC11 to POWER that adds a fence between relaxed reads and subsequent relaxed writes, but not between non-atomic accesses. As such, the only $\text{po} \cup \text{rf}$ cycles possible under the compilation scheme involve a racy non-atomic access. Since non-atomic races have undefined semantics in RC11, whenever there is such a cycle, the proof appeals to receptiveness to construct a different acyclic execution exhibiting the race.

The latter paper [19] introduced IMM and used it to establish correctness of compilation from the “promising semantics” (PS) [11] to the usual hardware models. As already mentioned, IMM’s definition catered precisely for the needs of the PS compilation proof, and so did not include important features such as sequentially consistent (SC) accesses. Our compilation proof shares some infrastructure with that proof – namely, the definition of IMM and traversals – but also has substantial differences because PS is quite different from *Weakestmo*. The main challenges in the PS proof were (1) to encode the various orders of the IMM execution graphs with the timestamps of the PS machine, and (2) to construct the certification runs for each outstanding promise. In contrast, the main technical challenge in the *Weakestmo* compilation proof is that event structures represent several possible executions of the program together, and that *Weakestmo* consistency includes constraints that correlate these executions, allowing one execution to affect the consistency of another.

7 Conclusion

In this paper, we presented the first correctness proof of mapping from the *Weakestmo* memory model to a number of hardware architectures. As a way to show correctness of *Weakestmo* compilation to hardware, we employed IMM [19], which we extended with SC accesses, from which compilation to hardware follows.

Although relying on IMM modularizes the compilation proof and makes it easy to extend to multiple architectures, it does have one limitation. As was discussed in Section 5.1.4, IMM enforces ordering between RMW events and subsequent memory accesses, while one desirable alternative compilation mapping of RMWs to ARMv8 does not enforce this ordering, which means that we cannot prove soundness of that mapping via the current definition of IMM. We are investigating whether one can weaken the corresponding IMM constraint, so that we can establish correctness of the alternative ARMv8 mapping as well.

Another way to establish correctness of this alternative mapping to ARMv8 may be to use the recently developed Promising-ARM model [22]. Indeed, since Promising-ARM is closely related to PS [11], it should be relatively easy to prove the correctness of compilation from

PS to Promising-ARM. Establishing compilation correctness of Weakestmo to Promising-ARM, however, would remain unresolved because Weakestmo and PS are incomparable [6]. Moreover, a direct compilation proof would probably also be quite difficult because of the rather different styles in which these models are defined.

References

- 1 Jade Alglave, Luc Maranget, and Michael Tautschnig. Herding cats: Modelling, simulation, testing, and data mining for weak memory. *ACM Trans. Program. Lang. Syst.*, 36(2):7:1–7:74, July 2014. doi:10.1145/2627752.
- 2 Mark Batty, Alastair F. Donaldson, and John Wickerson. Overhauling SC atomics in C11 and OpenCL. In *POPL 2016*, pages 634–648. ACM, 2016.
- 3 Mark Batty, Scott Owens, Susmit Sarkar, Peter Sewell, and Tjark Weber. Mathematizing C++ concurrency. In *POPL 2011*, pages 55–66, New York, 2011. ACM. doi:10.1145/1925844.1926394.
- 4 John Bender and Jens Palsberg. A formalization of java’s concurrent access modes. *Proc. ACM Program. Lang.*, 3(OOPSLA):142:1–142:28, October 2019. doi:10.1145/3360568.
- 5 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.
- 6 Soham Chakraborty and Viktor Vafeiadis. Grounding thin-air reads with event structures. *Proc. ACM Program. Lang.*, 3(POPL):70:1–70:27, 2019. doi:10.1145/3290383.
- 7 The Coq development of IMM, available at <http://github.com/weakmemory/imm>, 2019.
- 8 Stephen Dolan, KC Sivaramakrishnan, and Anil Madhavapeddy. Bounding data races in space and time. In *PLDI 2018*, pages 242–255, New York, 2018. ACM. doi:10.1145/3192366.3192421.
- 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, New York, 2016. ACM. doi:10.1145/2837614.2837615.
- 10 Alan Jeffrey and James Riely. On thin air reads towards an event structures model of relaxed memory. In *LICS 2016*, pages 759–767, New York, 2016. ACM. doi:10.1145/2933575.2934536.
- 11 Jeehoon Kang, Chung-Kil Hur, Ori Lahav, Viktor Vafeiadis, and Derek Dreyer. A promising semantics for relaxed-memory concurrency. In *POPL 2017*, pages 175–189, New York, 2017. ACM. doi:10.1145/3009837.3009850.
- 12 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.
- 13 Ori Lahav, Viktor Vafeiadis, Jeehoon Kang, Chung-Kil Hur, and Derek Dreyer. Repairing sequential consistency in C/C++11. In *PLDI 2017*, pages 618–632, New York, 2017. ACM. doi:10.1145/3062341.3062352.
- 14 Jeremy Manson, William Pugh, and Sarita V. Adve. The Java memory model. In *POPL 2005*, pages 378–391, New York, 2005. ACM. doi:10.1145/1040305.1040336.
- 15 C/C++11 mappings to processors, 2016. URL: <http://www.cl.cam.ac.uk/~pes20/cpp/cpp0xmappings.html>.
- 16 Evgenii Moiseenko, Anton Podkopaev, Ori Lahav, Orestis Melkonian, and Viktor Vafeiadis. Coq proof scripts and supplementary material for this paper, available at <http://plv.mpi-sws.org/weakestmoToImm/>, 2020.
- 17 Scott Owens, Susmit Sarkar, and Peter Sewell. A better x86 memory model: x86-TSO. In *TPHOLs 2009*, volume 5674 of *LNCS*, pages 391–407, Heidelberg, 2009. Springer.
- 18 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, New York, 2016. ACM. doi:10.1145/2837614.2837616.

5:26 Reconciling Event Structures with Modern Multiprocessors

- 19 Anton Podkopaev, Ori Lahav, and Viktor Vafeiadis. Bridging the gap between programming languages and hardware weak memory models. *Proc. ACM Program. Lang.*, 3(POPL):69:1–69:31, 2019. doi:10.1145/3290382.
- 20 Anton Podkopaev, Ilya Sergey, and Aleksandar Nanevski. Operational aspects of C/C++ concurrency. *CoRR*, abs/1606.01400, 2016. arXiv:1606.01400.
- 21 Christopher Pulte, Shaked Flur, Will Deacon, Jon French, Susmit Sarkar, and Peter Sewell. Simplifying ARM concurrency: multicopy-atomic axiomatic and operational models for ARMv8. *Proc. ACM Program. Lang.*, 2(POPL):19:1–19:29, 2018. doi:10.1145/3158107.
- 22 Christopher Pulte, Jean Pichon-Pharabod, Jeehoon Kang, Sung-Hwan Lee, and Chung-Kil Hur. Promising-ARM/RISC-V: a simpler and faster operational concurrency model. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2019*, pages 1–15, New York, NY, USA, 2019. ACM. doi:10.1145/3314221.3314624.
- 23 Jaroslav Ševčík, Viktor Vafeiadis, Francesco Zappa Nardelli, Suresh Jagannathan, and Peter Sewell. CompCertTSO: A verified compiler for relaxed-memory concurrency. *J. ACM*, 60(3):22, 2013. doi:10.1145/2487241.2487248.


Don't Panic! Better, Fewer, Syntax Errors for LR Parsers

Lukas Diekmann

Software Development Team, King's College London, United Kingdom

<https://lukasdiekmann.com/>

lukas.diekmann@gmail.com

Laurence Tratt 

Software Development Team, King's College London, United Kingdom

<https://tratt.net/laurie/>

laurie@tratt.net

Abstract

Syntax errors are generally easy to fix for humans, but not for parsers in general nor LR parsers in particular. Traditional “panic mode” error recovery, though easy to implement and applicable to any grammar, often leads to a cascading chain of errors that drown out the original. More advanced error recovery techniques suffer less from this problem but have seen little practical use because their typical performance was seen as poor, their worst case unbounded, and the repairs they reported arbitrary. In this paper we introduce the $CPCT^+$ algorithm, and an implementation of that algorithm, that address these issues. First, $CPCT^+$ reports the complete set of minimum cost repair sequences for a given location, allowing programmers to select the one that best fits their intention. Second, on a corpus of 200,000 real-world syntactically invalid Java programs, $CPCT^+$ is able to repair $98.37\% \pm 0.017\%$ of files within a timeout of 0.5s. Finally, $CPCT^+$ uses the complete set of minimum cost repair sequences to reduce the cascading error problem, where incorrect error recovery causes further spurious syntax errors to be identified. Across the test corpus, $CPCT^+$ reports $435,812 \pm 473$ error locations to the user, reducing the cascading error problem substantially relative to the $981,628 \pm 0$ error locations reported by panic mode.

2012 ACM Subject Classification Theory of computation → Parsing; Software and its engineering → Compilers

Keywords and phrases Parsing, error recovery, programming languages

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2020.6

Related Version Updates will be made available at <https://arxiv.org/abs/1804.07133>.

Supplementary Material ECOOP 2020 Artifact Evaluation approved artifact available at <https://doi.org/10.4230/DARTS.6.2.17>.

Funding This research was funded by the EPSRC Lecture (EP/L02344X/1) Fellowship.

Acknowledgements We are grateful to the Blackbox developers for allowing us access to their data, and particularly to Neil Brown for help in extracting a relevant corpus. We thank Edd Barrett for helping to set up our benchmarking machine and for comments on the paper. We also thank Carl Friedrich Bolz-Tereick, Sérgio Queiroz de Medeiros, Sarah Mount, François Pottier, Christian Urban, and Navaneetha Vasudevan for comments.

1 Introduction

Programming is a humbling job which requires acknowledging that we will make untold errors in our quest to perfect a program. Most troubling are semantic errors, where we intended the program to do one thing, but it does another. Less troubling, but often no less irritating, are syntax errors, which are generally minor deviances from the exacting syntax required



© Lukas Diekmann and Laurence Tratt;

licensed under Creative Commons License CC-BY

34th European Conference on Object-Oriented Programming (ECOOP 2020).

Editors: Robert Hirschfeld and Tobias Pape; Article No. 6; pp. 6:1–6:32

Leibniz International Proceedings in Informatics



LIPICs Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany



(a)	<pre>class C { int x y; }</pre>	(b)	<pre>C.java:2: error: ';' expected int x y; ^ C.java:2: error: <identifier> expected int x y; ^</pre>
(c)	<pre>Parsing error at line 2 col 9. Repair sequences found: 1: Delete y 2: Insert , 3: Insert =</pre>		

■ **Figure 1** An example of a simple, common Java syntax error (a) and the problems traditional error recovery has in dealing with it. `javac` (b) spots the error when it encounters “y”. Its error recovery algorithm then repairs the input by inserting a semicolon before “y” (i.e. making the input equivalent to “`int x; y;`”). This then causes a spurious parsing error, since “y” on its own is not a valid statement. The *CPCT*⁺ error recovery algorithm we introduce in this paper produces the output shown in (c): after spotting an error when parsing encounters “y”, it uses the Java grammar to find the complete set of minimum cost repair sequences (unlike previous approaches which non-deterministically find one minimum cost repair sequence). In this case three repair sequences are reported to the user: one can delete “y” entirely (“`int x;`”), or insert a comma (“`int x, y;`”), or insert an equals sign (“`int x = y;`”).

by a compiler. So common are syntax errors that parsers in modern compilers are designed to cope with us making several: rather than stop on the first syntax error, they attempt to *recover* from it. This allows them to report, and us to fix, all our syntax errors in one go.

When error recovery works well, it is a useful productivity gain. Unfortunately, most current error recovery approaches are simplistic. The most common grammar-neutral approach to error recovery are those algorithms described as “panic mode” (e.g. [13, p. 348]) which skip input until the parser finds something it is able to parse. A more grammar-specific variation of this idea is to skip input until a pre-determined synchronisation token (e.g. “;” in Java) is reached [8, p. 3], or to try inserting a single synchronisation token. Such strategies are often unsuccessful, leading to a cascade of spurious syntax errors (see Figure 1 for an example). Programmers quickly learn that only the location of the first error in a file – not the reported repair, nor the location of subsequent errors – can be relied upon to be accurate.

It is possible to hand-craft error recovery algorithms for a specific language. These generally allow better recovery from errors, but are challenging to create. For example, the Java error recovery approach in the Eclipse IDE is 5KLoC long, making it only slightly smaller than a modern version of Berkeley Yacc – a complete parsing system! Unsurprisingly, few real-world parsers contain effective hand-written error recovery algorithms.

Most of us are so used to these trade-offs (cheap generic algorithms and poor recovery vs. expensive hand-written algorithms and reasonable recovery) that we assume them to be inevitable. However, there is a long line of work on more advanced generic error recovery algorithms. Probably the earliest such algorithm is Aho and Peterson [1], which, upon encountering an error, creates on-the-fly an alternative (possibly ambiguous) grammar which allows the parser to recover. This algorithm has fallen out of favour in programming language circles, probably because of its implementation complexity and the difficulty of explaining to users what recovery has been used. A simpler family of algorithms, which trace their roots to Fischer et al. [11], instead try to find a single minimum cost *repair sequence* of token insertions and deletions which allow the parser to recover. Algorithms in this family are much better at recovering from errors than naive approaches and can communicate the repairs they find in a way that humans can easily replicate. However, such algorithms have seen little practical use because their typical performance is seen as poor and their worst case unbounded [17, p. 14]. We add a further complaint: such approaches only report a

single repair sequence to users. In general – and especially in syntactically rich languages – there are multiple reasonable repair sequences for a given error location, and the algorithm has no way of knowing which best matches the user’s intentions.

In this paper we introduce a new error recovery algorithm in the Fischer et al. family, *CPCT*⁺. This takes the approach of Corchuelo et al. [5] as a base, corrects it, expands it, and optimises its implementation. *CPCT*⁺ is simple to implement (under 500 lines of Rust code), is able to repair nearly all errors in reasonable time, reports the complete set of minimum cost repair sequences to users, and does so in less time than Corchuelo et al..

We validate *CPCT*⁺ on a corpus of 200,000 real, syntactically incorrect, Java programs (Section 6). *CPCT*⁺ is able to recover 98.37%±0.017% of files within a 0.5s timeout and does so while reporting fewer than half the error locations as a traditional panic mode algorithm: in other words, *CPCT*⁺ substantially reduces the cascading error problem. We also show – for, as far as we know, the first time – that advanced error recovery can be naturally added to a Yacc-esque system, allowing users to make fine-grained decisions about what to do when error recovery has been applied to an input (Section 7). We believe that this shows that algorithms such as *CPCT*⁺ are ready for wider usage, either on their own, or as part of a multi-phase recovery system.

1.1 Defining the problem

Formally speaking, we first test the following hypothesis:

H1 The complete set of minimum cost repair sequences can be found in acceptable time.

The only work we are aware of with a similar concept of “acceptable time” is [6], who define it as the total time spent in error recovery per file, with a threshold of 1s. We use that definition with one change: Since many compilers are able to fully execute in less than 1s, we felt that a tighter threshold is more appropriate: we use 0.5s since we think that even the most demanding users will tolerate such a delay. We strongly validate this hypothesis.

The complete set of minimum cost repair sequences makes it more likely that the programmer will see a repair sequence that matches their original intention (see Figure 1 for an example; Appendix A contains further examples in Java, Lua, and PHP). It also opens up a new opportunity. Previous error recovery algorithms find a single repair sequence, apply that to the input, and then continue parsing. While that repair sequence may have been a reasonable local choice, it may cause cascading errors later. Since we have the complete set of minimum cost repair sequences available, we can select from that set a repair sequence which causes fewer cascading errors. We thus rank repair sequences by how far they allow parsing to continue successfully (up to a threshold – parsing the whole file would, in general, be too costly), and choose from the subset that gets furthest (note that the time required to do this is included in the 0.5s timeout). We thus also test a second hypothesis:

H2 Ranking the complete set of minimum cost repair sequences by how far they allow parsing to continue locally reduces the global cascading error problem.

We also strongly validate this hypothesis. We do this by comparing “normal” *CPCT*⁺ with a simple variant *CPCT*_{rev}⁺ which reverses the ranking process, always selecting from amongst the worst performing minimum cost repair sequence. *CPCT*_{rev}⁺ models the worst case of previous approaches in the Fischer et al. family, which non-deterministically select a single minimum cost repair sequence. *CPCT*_{rev}⁺ leads to 31.93%±0.289% more errors being reported (i.e. it substantially worsens the cascading error problem).

This paper is structured as follows. We describe the Corchuelo et al. algorithm (Section 4), filling in missing details from the original description and correcting its definition. We then expand the algorithm into $CPCT^+$ (Section 5). Finally, we validate $CPCT^+$ on a corpus of 200,000 real, syntactically incorrect, Java programs comparing it to implementations of panic mode and Corchuelo et al. (Section 6). To emphasise that our algorithms are grammar-neutral, we show examples of error recovery on different grammars in Appendix A.

2 Background

We assume a high-level understanding of the mechanics of parsing in this paper, but in this section we provide a handful of definitions, and a brief refresher of relevant low-level details, needed to understand the rest of this paper. Although the parsing tool we created for this paper is written in Rust, we appreciate that this is still an unfamiliar language to many readers: algorithms are therefore given in Python which, we hope, is familiar to most.

Although there are many flavours of parsing, the Fischer et al. family of error recovery algorithms are designed to be used with $LR(k)$ parsers [16]. LR parsing remains one of the most widely used parsing approaches due to the ubiquity of Yacc [14] and its descendants (which include the Rust parsing tool we created for this paper). We use Yacc syntax throughout this paper so that examples can easily be tested in Yacc-compatible parsing tools.

Yacc-like tools take in a Context-Free Grammar (CFG) and produce a parser from it. The CFG has one or more *rules*; each rule has a name and one or more *productions* (often called “alternatives”); each production contains one or more *symbols*; and a symbol references either a *token type* or a grammar rule. One rule is designated the *start rule*. The resulting parser takes as input a stream of tokens, each of which has a *type* (e.g. INT) and a *value* (e.g. 123) – we assume the existence of a Lex-like tool which can split a string into a stream of tokens. Strictly speaking, parsing is the act of determining whether a stream of tokens is correct with respect to the underlying grammar. Since this is rarely useful on its own, Yacc-like tools allow grammars to specify “semantic actions” which are executed when a production in the grammar is successfully matched. Except where stated otherwise, we assume that the semantic actions build a *parse tree*, ordering the tokens into a tree of nonterminal nodes (which can have children) and terminal nodes (which cannot have children).

The CFG is first transformed into a *stategraph*, a statemachine where each node contains one or more *items* (describing the valid parse states at that point) and edges are labelled with terminals or nonterminals. Since even on a modern machine, a canonical (i.e. unmerged) LR stategraph can take several seconds to build, and a surprising amount of memory to store, we use the algorithm of [21] to merge together compatible states¹. The effect of this is significant, reducing the Java grammar we use later from 8908 to 1148 states. The stategraph is then transformed into a *statetable* with one row per state. Each row has a possibly empty *action* (shift, reduce, or accept) for each terminal and a possibly empty *goto state* for each nonterminal. Figure 2 shows an example grammar, its stategraph, and statetable.

The statetable allows us to define a simple, efficient, parsing process. We first define two functions relative to the statetable: $\text{action}(s, t)$ returns the action for the state s and token t or *error* if no such action exists; and $\text{goto}(s, N)$ returns the goto state for the state s and the nonterminal N or *error* if no such goto state exists. We then define a reduction relation

¹ [21] can over-merge states when conflicts occur [9, p. 3] (i.e. when Yacc uses precedence rules to turn an ambiguous grammar into an unambiguous LR parser). Since our error recovery approach operates purely on the statetable, it should work correctly with other merging approaches such as that of [9].

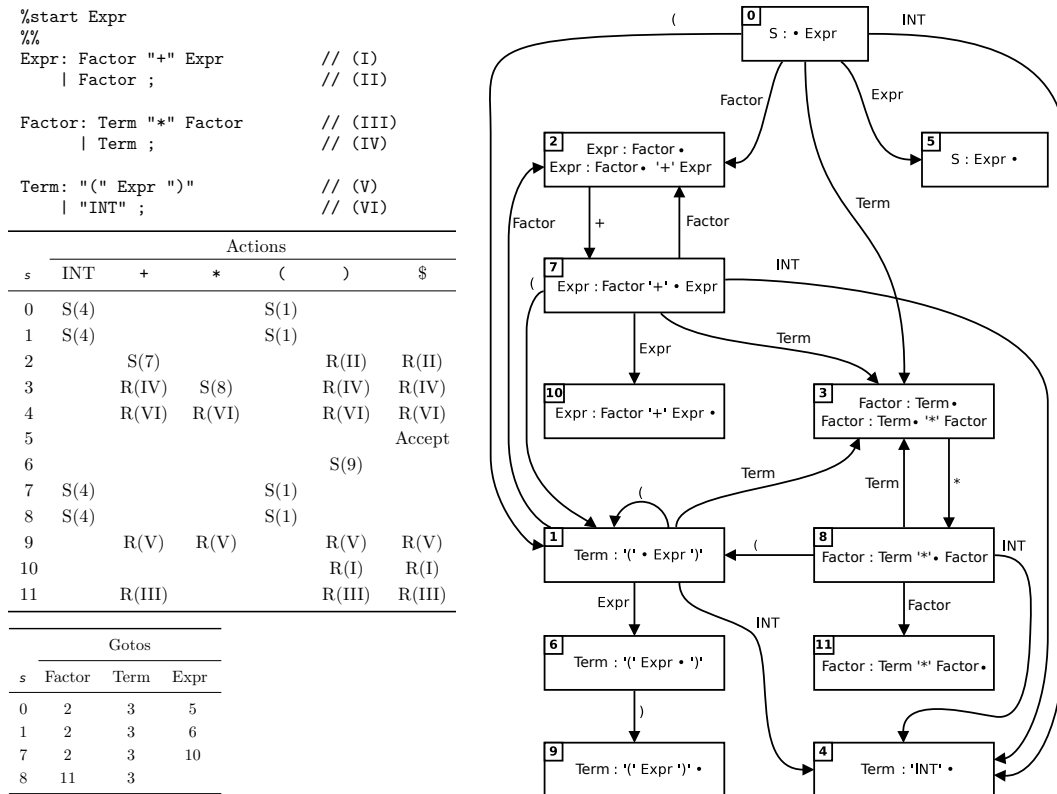


Figure 2 An example grammar (top left), its corresponding stategraph (right), and statetable (split into separate action and goto tables; bottom left). Productions in the grammar are labelled (I) to (VI). In the stategraph: $S(x)$ means “shift to state x ”; $R(x)$ means “reduce production x from the grammar” (e.g. $action(3, “+”)$ returns $R(IV)$ which references the production “**Factor**: **Term**;”). Each item within a state $[N : \alpha \bullet \beta]$ references one of rule N ’s productions; α and β each represent zero or more symbols; with the dot (\bullet) representing how much of the production must have been matched (α) if parsing has reached that state, and how much remains (β).

\rightarrow_{LR} for $(parsing\ stack, token\ list)$ pairs with two reduction rules as shown in Figure 3. A full LR parse \rightarrow_{LR}^* repeatedly applies the two \rightarrow_{LR} rules until neither applies, which means that $action(s_n, t_0)$ is either: *accept* (i.e. the input has been fully parsed); or *error* (i.e. an error has been detected at the terminal t_0). A full parse takes a starting pair of $([0], [t_0 \dots t_n, \$])$, where state 0 is expected to represent the entry point into the stategraph, $t_0 \dots t_n$ is the sequence of input tokens, and “\$” is the special End-Of-File (EOF) token.

3 Panic mode

Error recovery algorithms are invoked by a parser when it has yet to finish but there is no apparent way to continue parsing (i.e. when $action(s_n, t_0) = error$). Error recovery algorithms are thus called with a parsing stack and a sequence of remaining input (which we represent as a list of tokens): they can modify either or both of the parsing stack and the input in their quest to get parsing back on track. The differences between algorithms are thus in what modifications they can carry out (e.g. altering the parse stack; deleting input; inserting input), and how they carry such modifications out.

$$\frac{\text{action}(s_n, t_0) = \text{shift } s'}{([s_0 \dots s_n], [t_0 \dots t_n]) \rightarrow_{\text{LR}} ([s_0 \dots s_n, s'], [t_1 \dots t_n])} \text{ LR SHIFT}$$

$$\frac{(\text{action}(s_n, t_0) = \text{reduce } N : \alpha) \wedge (\text{goto}(s_{n-|\alpha|}, N) = s')}{([s_0 \dots s_n], [t_0 \dots t_n]) \rightarrow_{\text{LR}} ([s_0 \dots s_{n-|\alpha|}, s'], [t_0 \dots t_n])} \text{ LR REDUCE}$$

■ **Figure 3** Reduction rules for \rightarrow_{LR} , which operate on *(parsing stack, token list)* pairs. LR SHIFT advances the input by one token and grows the parsing stack, while LR REDUCE unwinds (“reduces”) the parsing stack when a production is complete before moving to a new (“goto”) state.

```

1 def holub(pstack, toks):
2     while len(toks) > 0:
3         npstack = pstack.copy()
4         while len(npstack) > 0:
5             if action(npstack[-1], toks[0]) != error: return (npstack, toks)
6             npstack.pop()
7         del toks[0]
8     return None

```

■ **Figure 4** Our version of the Holub [13] algorithm. This panic mode algorithm takes in a *(parsing stack, token list)* pair and returns: a *(parsing stack, token list)* pair if it managed to recover; or **None** if it failed to recover. The algorithm tries to find an element in the stack that has a non-error action for the next token in the input (lines 4–6). If it fails to find such an element, the input is advanced by one element (line 7) and the stack restored (line 3).

The simplest grammar-neutral error recovery algorithms are widely called “panic mode” algorithms (the origin of this family of algorithms seems lost in time). While there are several members of this family for LL parsing, there is only one fundamental way of creating a grammar-neutral panic mode algorithm for LR parsing. We take our formulation from Holub [13, p. 348]². The algorithm works by popping elements off the parsing stack to see if an earlier part of the stack is able to parse the next input symbol. If no element in the stack is capable of parsing the next input symbol, the next input symbol is skipped, the stack restored, and the process repeated. At worst, this algorithm guarantees to find a match at the EOF token. Figure 4 shows a more formal version of this algorithm.

The advantage of this algorithm is its simplicity and speed. For example, consider the grammar from Figure 2 and the input “2 + + 3”. The parser encounters an error on the second “+” token, leaving it with a parsing stack of [0, 2, 7] and the input “+ 3” remaining. The error recovery algorithm now starts. It first tries `action(7, “+”)` which (by definition, since it is the place the parser encountered an error) returns `error`; it then pops the top element from the parsing stack and tries `action(2, “+”)`, which returns `shift`. This is enough for the error recovery algorithm to complete, and parsing resumes with a stack [0, 2].

The fundamental problem with error recovery can be seen from the above example: by popping from the parsing stack, it implicitly deletes input from before the error location (in this case the first “+”) in order to find a way of parsing input after the error location. This often leads to panic mode throwing away huge portions of the input in its quest to find a repair. Not only can the resulting recovery appear as a *Deus ex machina*, but the more input that is skipped, the more likely that a cascade of further parsing errors ensues (as we will see later in Section 6.2).

² Note that step 2 in Holub causes valid repairs to be missed: while it is safe to ignore the top element of the parsing stack on the first iteration of the algorithm, as soon as one token is skipped, one must check all elements of the parsing stack. Our description simply drops step 2 entirely.

$$\begin{array}{c}
\frac{\text{action}(s_n, t) \neq \text{error} \wedge t \neq \$ \wedge ([s_0 \dots s_n], [t, t_0 \dots t_n]) \rightarrow_{\text{LR}}^* ([s'_0 \dots s'_m], [t_0 \dots t_n])}{([s_0 \dots s_n], [t_0 \dots t_n]) \rightarrow_{\text{CR}} ([s'_0 \dots s'_m], [t_0 \dots t_n], [\text{insert } t])} \text{ CR INSERT} \\
\\
\frac{t_0 \neq \$}{([s_0 \dots s_n], [t_0, t_1 \dots t_n]) \rightarrow_{\text{CR}} ([s_0 \dots s_n], [t_1 \dots t_n], [\text{delete}])} \text{ CR DELETE} \\
\\
\frac{([s_0 \dots s_n], [t_0 \dots t_n]) \rightarrow_{\text{LR}}^* ([s'_0 \dots s'_m], [t_j \dots t_n]) \wedge 0 < j \leq N_{\text{shifts}}}{j = N_{\text{shifts}} \vee \text{action}(s'_m, t_j) \in \{\text{accept}, \text{error}\}} \text{ CR SHIFT 1} \\
\frac{([s_0 \dots s_n], [t_0 \dots t_n]) \rightarrow_{\text{CR}} ([s'_0 \dots s'_m], [t_j \dots t_n], \underbrace{[\text{shift} \dots \text{shift}]_j})}{j}
\end{array}$$

■ **Figure 5** The repair-creating reduction rules for Corchuelo et al.. CR INSERT finds all terminals reachable from the current state and creates insert repairs for them (other than the EOF token “\$”). CR DELETE creates deletion repairs if user defined input remains. CR SHIFT 1 parses at least 1 and at most N_{shifts} tokens; if it reaches an accept or error state, or parses exactly N_{shifts} tokens, then a shift repair per token shifted is created.

4 Corchuelo et al.

There have been many attempts to create better LR error recovery algorithms than panic mode. Most numerous are those error recovery algorithms in what we call the Fischer et al. family. Indeed, there are far too many members of this family of algorithms to cover in one paper. We therefore start with one of the most recent – Corchuelo et al. [5]. We first explain the original algorithm (Section 4.1), although we use different notation than the original, fill in several missing details, and provide a more formal definition. We then make two correctness fixes to ensure that the algorithm always finds minimum cost repair sequences (Section 4.2). Since the original gives few details as to how the algorithm might best be implemented, we then explain our approach to making a fast implementation (Section 4.3).

4.1 The original algorithm

Intuitively, the Corchuelo et al. algorithm starts at the error state and tries to find a minimum cost repair sequence consisting of: *insert* T (“insert a token of type T ”), *delete* (“delete the token at the current offset”), or *shift* (“parse the token at the current offset”). The algorithm completes: successfully if it reaches an accept state or shifts “enough” tokens (N_{shifts} , set at 3 in Corchuelo et al.); or unsuccessfully if a repair sequence contains too many delete or insert repairs (set at 3 and 4 respectively in Corchuelo et al.) or spans “too much” input (N_{total} , set at 10 in Corchuelo et al.). Repair sequences are reported back to users with trailing *shift* repairs pruned i.e. $[\text{insert } x, \text{shift } y, \text{delete } z, \text{shift } a, \text{shift } b, \text{shift } c]$ is reported as $[\text{insert } x, \text{shift } y, \text{delete } z]$.

In order to find repair sequences, the algorithm keeps a breadth-first queue of *configurations*, each of which represents a different search state; configurations are queried for their neighbours which are put into the queue; and the search terminates when a successful configuration is found. The cost of a configuration is the sum of the repair costs in its repair sequence. By definition, a configuration’s neighbours have the same, or greater, cost to it.

As with the original, we explain the approach in two parts. First is a new reduction relation \rightarrow_{CR} which defines a configuration’s neighbours (Figure 5). Second is an algorithm which makes use of the \rightarrow_{CR} relation to generate neighbours, and determines when a successful configuration has been found or if error recovery has failed (Figure 6). As well as several

```

1 def corchueloetal(pstack, toks):
2     todo = [(pstack, toks, [])]
3     cur_cst = 0
4     while cur_cst < len(todo):
5         if len(todo[cur_cst]) == 0:
6             cur_cst += 1
7             continue
8         n = todo[cur_cst].pop()
9         if action(n[0][-1], n[1][0]) == accept or ends_in_N_shifts(n[2]): return n
10        elif len(n[1]) - len(toks) == N_total: continue
11        for nbr in all_cr_star(n[0], n[1]):
12            if len(n[2]) > 0 and n[2][-1] == delete and nbr[2][-1] == insert: continue
13            cst = cur_cst + rprs_cst(nbr[2])
14            for _ in range(len(todo), cst): todo.push([])
15            todo[cst].append((nbr[0], nbr[1], n[2] + nbr[2]))
16        return None
17
18 def rprs_cst(rprs):
19     c = 0
20     for r in rprs:
21         if r == shift: continue
22         c += 1
23     return c
24
25 def all_cr_star(pstack, toks):
26     # Exhaustively apply the  $\rightarrow_{CR}^*$  relation to
27     # (pstack, toks) and return the resulting
28     # list of (pstack, toks, repair) triples.

```

■ **Figure 6** Our version of the Corchuelo et al. algorithm. The main function `corchueloetal` takes in a (*parsing stack*, *token list*) pair and returns: a (*parsing stack*, *token list*, *repair sequence*) triple where *repair sequence* is guaranteed to be a minimum cost repair sequence; or `None` if it failed to find a repair sequence. The algorithm maintains a *todo* list of lists: the first sub-list contains configurations of cost 0, the second sub-list configurations of cost 1, and so on. The *todo* list is initialised with the error parsing stack, remaining tokens, and an empty repair sequence (line 2). If there are *todo* items left, a lowest cost configuration *n* is picked (lines 4–8). If *n* represents an accept state or if the last N_{shifts} repairs are shifts, then *n* represents a minimum cost repair sequence and the algorithm terminates successfully (line 9). If *n* has already consumed N_{total} tokens, then it is discarded (line 10). Otherwise, *n*'s neighbours are gathered using the \rightarrow_{CR} relation (lines 11, 25–28). To avoid duplicate repairs, *delete* repairs never follow *insert* repairs (line 12). Each neighbour has its repairs costed (line 13) and is then assigned to the correct *todo* sub-list (line 15). The `rprs_cst` function returns the cost of a repair sequence. Inserts and deletes cost 1, shifts 0.

changes for clarity, the biggest difference is that Figure 6 captures semi-formally what Corchuelo et al. explain in prose (spread amongst several topics over several pages): perhaps inevitably we have had to fill in several missing details. For example, Corchuelo et al. do not define what the cost of repairs is: for simplicities sake, we define the cost of *insert* and *delete* as 1, and *shift* as 0.³

4.2 Ensuring that minimum cost repair sequences aren't missed

CR SHIFT 1 (see Figure 5) has two flaws which prevent it from generating all possible minimum cost repair sequences.

First, CR SHIFT 1 always consumes input, missing intermediate configurations (including *accept* states!) that only require reductions/gotos to be performed. CR SHIFT 2 in Figure 7 shows the two-phase fix which addresses this problem. We first change the condition

³ It is trivial to extend this to variable token costs if desired, and our implementation supports this. However, it is unclear whether non-uniform token costs are useful in practise [4, p.96].

$$\frac{([s_0 \dots s_n], [t_0 \dots t_n]) \rightarrow_{LR}^* ([s'_0 \dots s'_m], [t_j \dots t_n]) \wedge 0 \leq j \leq N_{shifts} \quad (j = 0 \wedge [s_0 \dots s_n] \neq [s'_0 \dots s'_m]) \vee j = N_{shifts} \vee \text{action}(s'_m, t_j) \in \{\text{accept}, \text{error}\}}{([s_0 \dots s_n], [t_0 \dots t_n]) \rightarrow_{CR} ([s'_0 \dots s'_m], [t_j \dots t_n], \underbrace{[shift \dots shift]}_j)} \quad \text{CR SHIFT 2}$$

$$\frac{([s_0 \dots s_n], [t_0 \dots t_n]) \rightarrow_{LR}^* ([s'_0 \dots s'_m], [t_j \dots t_n]) \wedge 0 \leq j \leq 1 \quad (j = 0 \wedge [s_0 \dots s_n] \neq [s'_0 \dots s'_m] \wedge R = []) \vee (j = 1 \wedge R = [shift])}{([s_0 \dots s_n], [t_0 \dots t_n]) \rightarrow_{CR} ([s'_0 \dots s'_m], [t_j \dots t_n], R)} \quad \text{CR SHIFT 3}$$

■ **Figure 7** CR SHIFT 1 always consumes input, when sometimes performing one or more reduction/gotos without consuming input would be better. CR SHIFT 2 addresses this issue. Both CR SHIFT 1 and CR SHIFT 2 generate multiple shift repairs in one go, which causes them to skip “intermediate” (and sometimes important) configurations. CR SHIFT 3 generates at most one shift, exploring all intermediate configurations.

(a)	Delete 3, Delete + Delete 3, Shift +, Insert Int Insert +, Shift 3, Shift +, Insert Int Insert *, Shift 3, Shift +, Insert Int	(b)	Insert *, Shift 3, Delete + Insert +, Shift 3, Delete +
-----	---	-----	--

■ **Figure 8** Given the input “2 3 +” and the grammar from Figure 2, CR SHIFT 1 is unable to find any repair sequences because it does not perform the reductions/gotos necessary after the final *insert* or *delete* repairs to reach an accept state. (a) CR SHIFT 2 can find 4 minimum cost repair sequences. (b) CR SHIFT 3 can find a further 2 minimum cost repair sequences on top those found by CR SHIFT 2 (i.e. 6 in total).

$0 < j \leq N_{shifts}$ to $0 \leq j \leq N_{shifts}$ so that the parser can make progress without consuming input. However, this opens the possibility of an infinite loop, so we then add a condition that if no input is consumed, the parsing stack must have changed. In other words, we require progress to be made, whether or not that progress involved consuming input.

Second, CR SHIFT 1 and CR SHIFT 2 generate multiple shifts at a time. This causes them to skip intermediate configurations from which minimum cost repair sequences may be found. The solution⁴ is simple: at most one shift can be generated at any one time. CR SHIFT 3 in Figure 7 (as well as incorporating the fix from CR SHIFT 2) generates at most one shift repair at a time. Relative to CR SHIFT 1, it is simpler, though it also inevitably slows down the search, as more configurations are generated.

The problems with CR SHIFT 1, in particular, can be severe. Figure 8 shows an example input where CR SHIFT 1 is unable to find any repair sequences, CR SHIFT 2 some, and CR SHIFT 3 all minimum cost repair sequences.

4.3 Implementation considerations

The definitions we have given thus far do not obviously lead to an efficient implementation and Corchuelo et al. give few useful hints. We found that two techniques were both effective at improving performance while being simple to implement.

⁴ The problem, and the basis of a fix, derive from [15, p. 12], though their suggestion suffers from the same problem as CR SHIFT 1.

First, Corchuelo et al. suggest using a breadth-first search but give no further details. It was clear to us that the most natural way to model the search is as an instance of Dijkstra's algorithm. However, rather than use a general queue data-structure (probably based on a tree) to discover which element to search next, we use a similar queue data-structure to [4, p. 25]. This consists of one sub-list per cost (i.e. the first sub-list contains configurations of cost 0, the second sub-list configurations of cost 1 and so on). Since we always know what cost we are currently investigating, finding the next todo element requires only a single pop (line 8 of Figure 6). Similarly, adding elements requires only an `append` to the relevant sub-list (lines 18, 21, 22). This data-structure is a good fit because costs in our setting are always small (double digits is unusual for real-world grammars) and each neighbour generated from a configuration with cost c has a cost $\geq c$.

Second, since error recovery frequently adjusts and resets parsing stacks and repair sequences, during error recovery we do not represent these as lists (as is the case during normal parsing). We found that lists consume noticeably more memory, and are slightly less efficient, than using parent pointer trees (often called “cactuses”). Every node in such a tree has a reference to a single parent (or `null` for the root node) but no references to child nodes. Since our implementation is written in Rust – a language without garbage collection – nodes are reference counted (i.e. a parent is only freed when it is not in a todo list and no children point to it). When the error recovery algorithm starts, it converts the main parsing stack (a list) into a parent pointer tree; and repair sequences start as empty parent pointer trees. The \rightarrow_{CR} part of our implementation thus operates exclusively on parent pointer trees. Although this does mean that neighbouring configurations are scattered throughout memory, the memory sharing involved seems to more than compensate for poor cache behaviour; it also seems to be a good fit with modern `malloc` implementations, which are particularly efficient when allocating and freeing objects of the same size. This representation is likely to be a reasonable choice in most contexts, although it is difficult to know from our experience whether it will always be the best choice (e.g for garbage collected languages).

One seemingly obvious improvement is to split the search into parallel threads. However, we found that the nature of the problem means that parallelisation is more tricky, and less productive, than might be expected. There are two related problems: we cannot tell in advance if a given configuration will have huge numbers of successors or none at all; and configurations are, in general, searched for successors extremely quickly. Thus if we attempt to seed threads with initial sets of configurations, some threads quickly run out of work whilst others have ever growing queues. If, alternatively, we have a single global queue then significant amounts of time can be spent adding or removing configurations in a thread-safe manner. A work stealing algorithm might solve this problem but, as we shall see in Section 6, *CPCT+* runs fast enough that the additional complexity of such an approach is not, in our opinion, justified.

5 *CPCT+*

In this section, we extend the Corchuelo et al. algorithm to become what we call *CPCT+*. First we enhance the algorithm to find the complete set of minimum cost repair sequences (Section 5.1). Since this slows down the search, we optimise by merging together compatible configurations (Section 5.2). The complete set of minimum cost repair sequences allows us to make an algorithm less susceptible to the cascading error problem (Section 5.3). We then change the criteria for terminating error recovery (Section 5.4).

5.1 Finding the complete set of minimum cost repair sequences

The basic Corchuelo et al. algorithm non-deterministically completes as soon as it has found a single minimum cost repair sequence. This is confusing in two different ways: the successful repair sequence found can vary from run to run; and the successful repair sequence might not match the user’s intention.

We therefore introduce the idea of the complete set of minimum cost repair sequences: that is, all equivalently good repair sequences. Although we will refine the concept of “equivalently good” in Section 5.3, at this stage we consider all successful repair sequences with minimum cost c to be equivalently good. In other words, as soon as we find the first successful repair sequence, its cost c defines the minimum cost.

An algorithm to generate this set is then simple: when a repair sequence of cost c is found to be successful, we discard all repair sequences with cost $> c$, and continue exploring configurations in cost c (including, transitively, all neighbours that are also of cost c ; those with cost $> c$ are immediately discarded). Each successful configuration is recorded and, when all configurations in c have been explored, the set of successful configurations is returned. One of these successful configurations is then non-deterministically chosen, applied to the input, and parsing continued.

5.2 Merging compatible configurations

Relative to finding a single solution, finding the complete set of repair sequences can be extremely expensive because there may be many remaining configurations in c , which may, transitively, have many neighbours. Our solution to this performance problem is to merge together *compatible* configurations on-the-fly, preserving their distinct repair sequences while reducing the search space. Two configurations are compatible if:

1. their parsing stacks are identical,
2. they both have an identical amount of input remaining,
3. and their repair sequences are compatible.

Two repair sequences are compatible:

1. if they both end in the same number ($n \geq 0$) of shifts,
2. and, if one repair sequence ends in a delete, the other repair sequence also ends in a delete.

The first of these conditions is a direct consequence of the fact that a configuration is deemed successful if it ends in N_{shifts} shift repairs. When we merge configurations, one part of the merge is “dominant” (i.e. checked for N_{shifts}) and the other “subsumed”: we have to maintain symmetry between the two to prevent the dominant part accidentally preventing the subsumed part from being recorded as successful. In other words, if the dominant part of the merge had fewer shifts at the end of its repair sequence than the subsumed part, then the N_{shifts} check (line 10, Figure 6) would fail, even though reversing the dominant and subsumed parts may have lead to success. It is therefore only safe to merge repair sequences which end in the same number of shifts.

The second condition relates to the weak form of compatible merging inherited from [5, p. 8]: delete repairs are never followed by an insert (see Figure 6) since $[delete, insert\ x]$ always leads to the same configuration as $[insert\ x, delete]$. Although we get much of the same effect through compatible configuration merging, we keep it as a separate optimisation because: it is such a frequent case; our use of the todo list means that we would not catch

every case; the duplicate repair sequences are uninteresting from a user perspective, so we would have to filter them out later anyway; and each additional merge costs memory. We thus have to make sure that merged repair sequences don't accidentally suppress insert repairs because one part of the repair sequence ends in a delete while the other does not. The simplest way of solving this problem is thus to forbid merging repair sequences if one sequence ends in a delete and the other does not.

Fortunately, implementing compatible configuration merging is simple. We first modify the `todo` data-structure to be a list-of-ordered-hashsets⁵. This has near-identical `append` / `pop` performance to a normal list, but filters out duplicates with near-identical performance to an unordered hashset. We then make use of a simple property of hashsets: an object's hashing behaviour need only be a non-strict subset of its equality behaviour. Configuration hashing is based solely on a configuration's parsing stack and remaining input, giving us a fast way of finding configurations that are compatible under conditions #1 (identical parsing stacks) and #2 (identical input remaining). As well as checking those two conditions, configuration equality also checks condition #3 (compatible repair sequences).

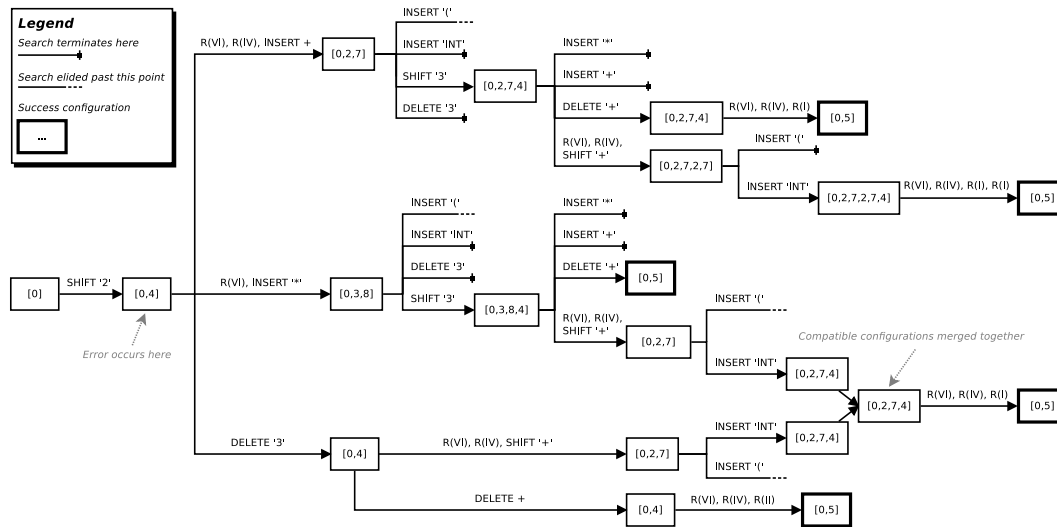
Conceptually, merging two configurations together is simple: each configuration needs to store a set of repair sequences, each of which is updated as further repairs are found. However, this is an extremely inefficient representation as the sets involved need to be copied and extended as each new repair is found. Instead, we reuse the idea of graph-structured stacks from GLR parsing [28, p. 4] which allows us to avoid copying whenever possible. The basic idea is that configurations no longer reference a parent pointer tree of repairs directly, but instead a parent pointer tree of *repair merges*. A repair merge is a pair (*repair*, *merged*) where *repair* is a plain repair and *merged* is a (possibly null) set of repair merge sequences. This structure has two advantages. First, the N_{shifts} check can be performed solely using the first element of repair merge pairs. Second, we avoid allocating memory for configurations which have not yet been subject to a merge. The small downside to this scheme is that expanding configurations into repair sequences requires recursively expanding both the normal parent pointer tree of the first element as well as the merged parent pointer trees of the second element.

Compatible configuration merging is particularly effective in complex cases, even though it can only merge configurations in the `todo` list (i.e. we cannot detect all possible compatible merges). An example of compatible configuration merging can be seen in Figure 9.

5.3 Ranking repair sequences

In nearly all cases, members of the complete set of minimum cost repair sequences end with N_{shifts} (the only exception being error locations near the end of an input where recovery leads to an accept state). Thus while the repair sequences we find are all equivalently good within the range of N_{shifts} , some, but not others, may perform poorly beyond that range. This problem is exacerbated by the fact that N_{shifts} has to be a fairly small integer (we use 3, the value suggested by Corchuelo et al.) since each additional token searched exponentially increases the search space. Thus while all repair sequences found may be locally equivalent, when considered in the context of the entire input, some may be better than others. While it is, in general, impractically slow to determine which repair sequences are the global best, we can quickly determine which are better under a wider definition of "local".

⁵ An ordered hashset preserves insertion order, and thus allows list-like integer indexing as well as hash-based lookups.



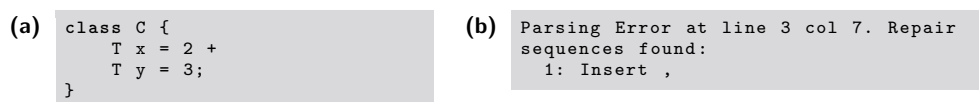
■ **Figure 9** An elided visualisation of a real run of $CPCT^+$ with the input “2 3 +” and the grammar from Figure 2. The left hand side of the tree shows the “normal” parser at work, which hits an error as soon as it has shifted the token “2”: at this point, $CPCT^+$ starts operating. As this shows, the search encounters various dead ends, as well as successful routes. As shown in Figure 8, this input has 6 minimum cost repair sequences, but the search only has 5 success configurations, because two configurations were merged together.

We thus rank configurations which represent the complete set of minimum cost repair sequences by how far they allow parsing to continue, up to a limit of N_{try} tokens (which we somewhat arbitrarily set at 250). Taking the furthest-parse point as our top rank, we then discard all configurations which parsed less input than this. The reason why we rank the configurations, and not the repair sequences, is that we only need to rank one of the repair sequences from each merged configuration, a small but useful optimisation. We then expand the top ranked configurations into repair sequences and remove shifts from the end of those repair sequences. Since the earlier merging of compatible configurations is imprecise (it misses configurations that have already been processed), there can be some remaining duplicate repair sequences: we thus perform a final purge of duplicate repair sequences. Figure 9 shows a visualisation of $CPCT^+$ in action.

Particularly on real-world grammars, selecting the top-ranked repair sequences substantially decreases cascading errors (see Figure 10 for an example). It also does so for very little additional computational cost, as the complete set of minimum cost repair sequences typically contains only a small number of items. However, it cannot entirely reduce the cascading error problem. Since, from our perspective, each member of the top-ranked set is equivalently good, we non-deterministically select one of its members to repair the input and allow parsing to continue. This can mean that we select a repair sequence which performs less well beyond N_{try} tokens than other repair sequences in the top-ranked set.

5.4 Timeout

The final part of $CPCT^+$ relates to the use of N_{total} in Corchuelo et al.. As with all members of the Fischer et al. family, $CPCT^+$ is not only unbounded in time [17, p. 14], but also unbounded in memory. In an attempt to combat this, Corchuelo et al. limits repair sequences to a maximum of 3 deletes and 4 inserts and a span of at most 10 tokens, attempting to stop



■ **Figure 10** An example showing how the ranking of repair sequences can lessen the cascading error problem. The Java example (a) leads to a parsing error on line 3 at “y”, with three minimum cost repair sequences found: `[insert ,]`, `[insert ?]`, and `[insert (]`. These repair sequences are then ranked by how far they allow parsing to continue successfully. `[insert ,]` leads to the rest of the file being parsed without further error. `[insert ?]` causes a cascading error at “;” which must then be resolved by completing the ternary expression started by “?” (e.g. changing line 3 to “T ? y : this;”). Similarly, `[insert (]` causes a cascading error at “;” which must then be resolved by inserting a “)”. Since `[insert ,]` is ranked more highly than the other repair sequences, the latter are discarded, leading to the parsing output shown in (b). `javac` in contrast attempts to insert “;” before “y” causing a cascading error on the next token.

the search from going too far. Unfortunately it is impossible to find good values for these constants, as “too far” is entirely dependent on the grammar and erroneous input: Java’s grammar, for example, is large with a commensurately large search space (requiring smaller constants) while Lua’s grammar is small with a commensurately small search space (which can cope with larger constants).

This problem can be easily seen on inputs with unbalanced brackets (e.g. expressions such as “x = f(();”): each additional unmatched bracket exponentially increases the search space. On a modern machine with a Java 7 grammar, *CPCT*⁺ takes about 0.3s to find the complete set of minimum cost repair sequences for 3 unmatched brackets, 3s for 4 unmatched brackets, and 6 unmatched brackets caused our 32GiB test machine to run out of RAM.

The only sensible alternative is a timeout: up to several seconds is safe in our experience. We thus remove N_{total} from *CPCT*⁺ and rely entirely on a timeout which, in this paper, is defined to be 0.5s.

6 Experiment

In order to understand the performance of *CPCT*⁺, we conducted a large experiment on real-world Java code. In this section we outline our methodology (Section 6.1) and results (Section 6.2). Our experiment is fully repeatable and downloadable from https://archive.org/download/error_recovery_experiment/0.4/. The results from our particular run of the experiment can also be downloaded from the same location.

6.1 Methodology

In order to evaluate error recovery implementations, we need a concrete implementation. We created a new Yacc-compatible parsing system *grmtools* in Rust which we use for our experiments. Including associated libraries for LR table generation and so on, *grmtools* is around 13KLoC. Although intended as a production library, it has accidentally played a part as a flexible test bed for experimenting with, and understanding, error recovery algorithms. We added a simple front-end *nimbleparse* which produces the output seen in e.g. Figure 1.

There are two standard problems when evaluating error recovery algorithms: how to determine if a good job has been done on an individual example; and how to obtain sufficient examples to get a wide perspective on an algorithm’s performance. Unfortunately, solutions to these problems are mutually exclusive, since the only way to tell if a good job has been

done on a particular is to manually evaluate it, which means that it is only practical to use a small set of input programs. Most papers we are aware of use at most 200 source files (e.g. [5]), with one using a single source file with minor variants [15]. [4] was the first to use a large-scale corpus of approximately 60,000 Java source files. Early in the development of our methodology, we performed some rough experiments which suggested that statistics only start to stabilise once a corpus exceeds 10,000 source files. We therefore prefer to use a much larger corpus than most previous studies. We are fortunate to have access to the Blackbox project [3], an opt-in data collection facility for the BlueJ editor, which records major editing events (e.g. compiling a file) and sends them to a central repository. Crucially, one can see the source code associated with each event. What makes Blackbox most appealing as a data source is its scale and diversity: it has hundreds of thousands of users, and a huge collection of source code.

We first obtained a Java 1.5 Yacc grammar and updated it to support Java 1.7.⁶ We then randomly selected source files from Blackbox’s database (following the lead of [27], we selected data from Blackbox’s beginning until the end of 2017-12-31). We then ran such source files through our Java 1.7 lexer. We immediately rejected files which didn’t lex, to ensure we were dealing solely with parsing errors⁷ (see Section 7.4). We then parsed candidate files with our Java grammar and rejected any which did parse successfully, since there is little point running an error recovery algorithm on correct input. The final corpus consists of 200,000 source files (collectively a total of 401MiB). Since Blackbox, quite reasonably, requires each person with access to the source files to register with them, we cannot distribute the source files directly; instead, we distribute the (inherently anonymised) identifiers necessary to extract the source files for those who register with Blackbox.

The size of our corpus means that we cannot manually evaluate repairs for quality. Instead, we report several other metrics, of which the number of error locations is perhaps the closest proxy for perceived quality. However, this number has to be treated with caution for two reasons. First, it is affected by differences in the failure rate: if a particular error recovery algorithm cannot repair an entire file then it may not have had time to find all the “true” error locations. Second, the number of error locations only allows relative comparisons. Although we know that the corpus contains at least 200,000 manually created errors (i.e. at least one per file), we cannot know if, or how many, files contain more than one error. Since we cannot know the true number of error locations, we are unable to evaluate algorithms in an absolute sense.

In order to test hypothesis H1 we ran each error recovery algorithm against the entire Java corpus, collecting for each file: the time spent in recovery (in seconds); whether error recovery on the file succeeded or failed (where failure is due to either the timeout being exceeded or no repair sequences being found for an error location); the number of error locations; the cost of repair sequences at each error location; and the proportion of tokens skipped by error recovery (i.e. how many *delete* repairs were applied). We measure the time spent in error recovery with a monotonic wall-clock timer, covering the time from when the main parser first invokes error recovery until an updated parsing stack and parsing index are returned along with minimum cost repair sequences. The timer is suspended when normal parsing restarts and resumed if error recovery is needed again (i.e. the timeout applies to the file as a whole).

⁶ Unfortunately, changes to the method calling syntax in Java 1.8 mean that it is an awkward, though not impossible, fit for an LR(1) formalism such as Yacc, requiring substantial changes to the current Java Yacc grammar. We consider the work involved beyond that useful for this paper.

⁷ Happily, this also excludes files which can’t possibly be Java source code. Some odd things are pasted into text editors.

We evaluate three main error recovery algorithms: Corchuelo et al., $CPCT^+$, and panic mode. Our implementation of Corchuelo et al. is to some extent a “best effort” since we have had to fill in several implementation details ourselves. As per the description, we: use the same limits on repair sequences (repair sequences can contain at most 3 delete or 4 insert repairs, and cannot span more than 10 tokens in the input); complete as soon as a single successful repair sequence is found; and, when no available repair sequence is found, fall back on panic mode. In addition, we impose the same 0.5s timeout on this algorithm, as it is otherwise unbounded in length, and sometimes exhausts available RAM. Panic mode implements the algorithm from Section 3. We do not report the average cost size for panic mode or Corchuelo et al. (which falls back on panic mode) since they do not (always) report repair sequences (see Section 3). Although panic mode can implicitly delete input from before the error location, we only include the input it explicitly skips in the proportion of tokens skipped.

In order to test hypothesis H2, we created a variant of $CPCT^+$ called $CPCT_{rev}^+$. Instead of selecting from the minimum cost repair sequences which allow parsing to continue furthest, $CPCT_{rev}^+$ selects from those which allow parsing to continue the least far. This models the worst case for other members of the Fischer et al. family which non-deterministically select a single minimum cost repair sequence. In other words, it allows us to understand how many more errors could be reported to users of other members of the Fischer et al. family compared to $CPCT^+$.

Configuration merging (see Section 5.2) is the most complex part of $CPCT^+$. To understand whether this complexity leads to better performance, we created another variant of $CPCT^+$ called $CPCT_{DM}^+$ which disables configuration merging.

We bootstrap [10] our results 10,000 times to produce 99% confidence intervals. However, as Figure 12 shows, our distribution is heavy-tailed, so we cannot bootstrap naively. Instead, we ran each error recovery algorithm 30 times on each source file; when bootstrapping we randomly sample one of the 30 values collected (i.e. our bootstrapped data contains an entry for every file in the experiment; that entry is one of the 30 values collected for that file).

All experiments were run on an otherwise unloaded Intel Xeon E3-1240 v6 with 32GiB RAM running Debian 10. We disabled hyperthreading and turbo boost and ran experiments serially. Our experiments took approximately 15 days to complete. We used Rust 1.43.1 to compile *grmtools* (the `Cargo.lock` file necessary to reproduce the build is included in our experimental repository).

6.2 Results

Figure 11 shows a summary of the results of our experiment. Comparing the different algorithms requires care as a higher failure rate tends to lower the cost size, tokens skipped, and number of error locations simply because files are not completely parsed. For example, although Corchuelo et al. reports fewer error locations than $CPCT^+$, that is probably due to Corchuelo et al.’s higher failure rate; however, as we shall see in Section 6.3, panic mode’s much higher number of error locations relative to $CPCT^+$ might better be explained by other factors.

With that caution in mind, the overall conclusions are fairly clear. $CPCT^+$ is able to repair nearly all input files within the 0.5s timeout. While panic mode is able to repair every file within the 0.5s timeout, it reports well over twice as many error locations as $CPCT^+$ —in other words, panic mode substantially worsens the cascading error problem. As well as producing more detailed and accurate output, $CPCT^+$ has a lower failure rate, median, and mean time than Corchuelo et al.. The fact that the median recovery time for $CPCT^+$ is two

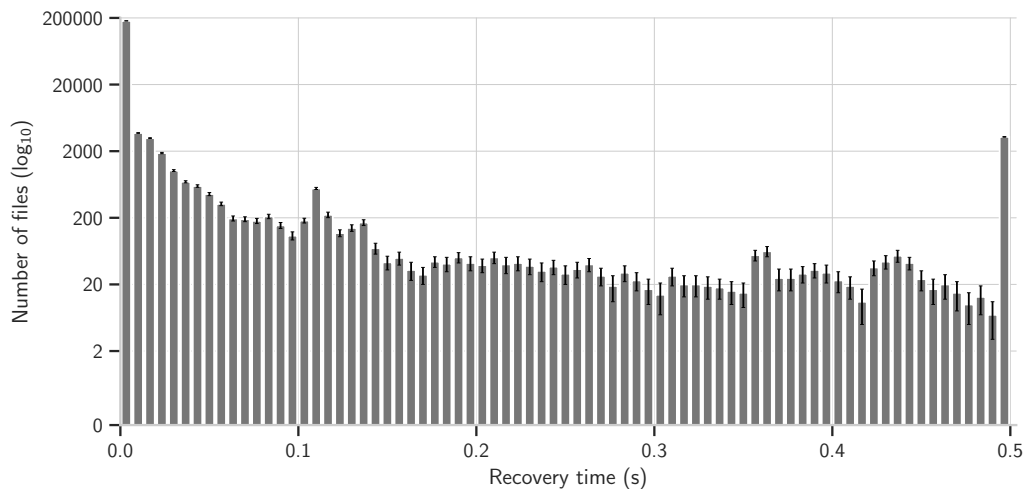
	Mean time (s)	Median time (s)	Cost size (#)	Failure rate (%)	Tokens skipped (%)	Error locations (#)
Corchuelo et al.	0.042367 ± 0.0000057	0.000335 ± 0.0000007	-	5.54 ± 0.004	0.61 $\pm < 0.001$	374,731 ± 26
$CPCT^+$	0.013643 ± 0.0000822	0.000251 ± 0.0000003	1.67 ± 0.001	1.63 ± 0.017	0.31 ± 0.001	435,812 ± 473
Panic mode	0.000002 $\pm < 0.0000001$	0.000001 $\pm < 0.0000001$	-	0.00 $\pm < 0.001$	3.72 $\pm < 0.001$	981,628 ± 0
$CPCT^+_{DM}$	0.026127 ± 0.0001077	0.000258 ± 0.0000003	1.63 ± 0.001	3.63 ± 0.025	0.28 ± 0.001	421,897 ± 358
$CPCT^+_{rev}$	0.018374 ± 0.0001109	0.000314 ± 0.0000006	1.77 ± 0.001	2.34 ± 0.023	0.41 ± 0.002	574,979 ± 1104

■ **Figure 11** Summary statistics from running various error recovery algorithms over a corpus of 200,000 Java files (for all measures, lower is better). Mean and median times report how much time was spent in error recovery per file: both figures include files which exceeded the recovery timeout, so they represent the “real” times that users would experience, whether or not all errors are repaired or not. Cost size reports the mean cost (i.e. the number of insert and delete repairs) of each error location repaired (this number is meaningless for Corchuelo et al. – which falls back on panic mode – and panic mode, since panic mode does not report repair sequences). The failure rate is the percentage of files which could not be fully repaired within the timeout. Tokens skipped is the proportion of input skipped (because of a delete repair). $CPCT^+_{rev}$ models the worst case of non-deterministic Fischer et al. algorithms by reversing the order of repair ranking (see Section 5.3). $CPCT^+_{DM}$ shows the performance of $CPCT^+$ if configuration merging is disabled (see Section 5.2).

orders of magnitude lower than its mean recovery time suggests that only a small number of outliers cause error recovery to take long enough to be perceptible to humans; this is confirmed by the histogram in Figure 12. These results strongly validate Hypothesis H1.

Corchuelo et al.’s poor performance may be surprising, as it produces at most one (possibly non-minimum cost) repair sequence whereas $CPCT^+$ produces the complete set of minimum cost repair sequences – in other words, $CPCT^+$ is doing more work, more accurately, and in less time than Corchuelo et al.. There are three main reasons for Corchuelo et al.’s poor performance. First, the use of CR SHIFT 1 causes the search to miss intermediate nodes that would lead to success being detected earlier. Second, the heuristics used to stop the search from going too far (e.g. limiting a repair sequence’s number of inserts and deletes) are not well-suited to a large grammar such as Java’s: the main part of the search often exceeds the timeout, leaving no time for the fallback mechanism of panic mode to be used. Finally, Corchuelo et al. lacks configuration merging, causing it to perform needless duplicate work.

$CPCT^+$ ranks the complete set of minimum cost repair sequences by how far parsing can continue and chooses from those which allow parsing to continue furthest. $CPCT^+_{rev}$, in contrast, selects from those which allow parsing to continue the least far. $CPCT^+_{rev}$ shows that the ranking technique used in $CPCT^+$ substantially reduces the potential for cascading errors: $CPCT^+_{rev}$ leads to $31.93\% \pm 0.289\%$ more error locations being reported to users relative to $CPCT^+$. We visualise this in the histogram of Figure 13 which shows all files with 1–50 error locations (a complete histogram can be found in Figure 18 in the Appendix). Note that files where error recovery did not complete and no error locations were found (which happens occasionally with $CPCT^+_{rev}$) are excluded from this histogram (since we know that every file in the corpus has at least one error), but files where error recovery did not complete but some error locations were found are included (since this gives us, at least, a lower bound on the number of error locations). As Figure 13 shows, the distribution of error locations in



■ **Figure 12** A histogram of the time spent in error recovery by $CPCT^+$ for files in our corpus. The x axis shows time (up to the timeout of 0.5s) and the y axis is a logarithmic scale for the number of files. Error bars represent 99% confidence intervals. As this clearly shows, the vast majority of files fit in the histogram's first bin; there is then a gradual decrease until around 0.15s, with a broadly flat distribution from then until the pronounced peak at the timeout of 0.5s. Figure 17 in the Appendix shows how extending the timeout increases the number of files which can be successfully recovered.

$CPCT^+$ and $CPCT_{rev}^+$ is similar, with the latter simply shifted slightly to the right. In other words, $CPCT_{rev}^+$ makes error recovery slightly worse in a number of files (rather than making error recovery in a small number of files a lot worse). This strongly validates Hypothesis H2.

Interestingly, and despite its higher failure rate, $CPCT_{rev}^+$ has a noticeably higher mean cost of repair sequences relative to $CPCT^+$. In other words, $CPCT_{rev}^+$ not only causes more error locations to be reported, but those additional error locations have longer repair sequences. This suggests that there is a double whammy from cascading errors: not only are more error locations reported, but the poorer quality repair sequences chosen make subsequent error locations disproportionately harder for the error recovery algorithm to recover from.

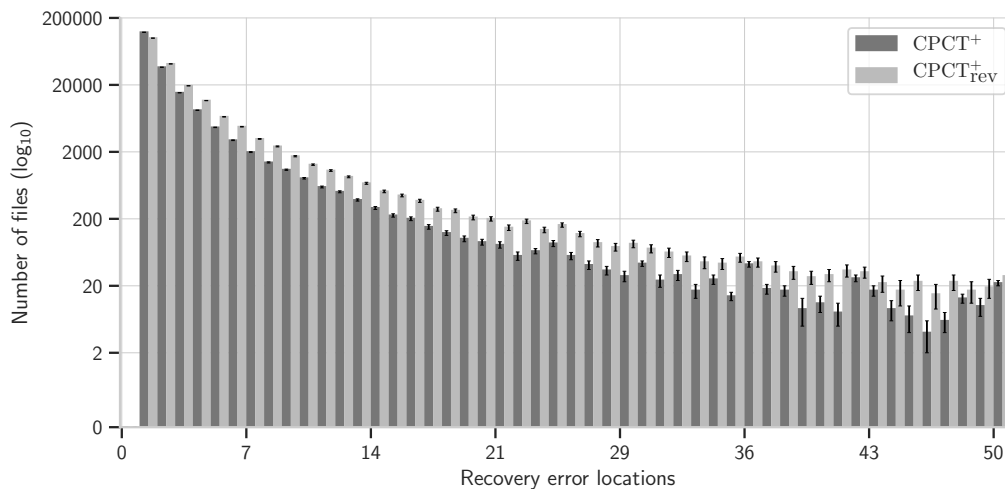
$CPCT_{DM}^+$ shows that configuration merging has a significant effect on the failure rate, in our opinion justifying both its conceptual complexity and the less than 100LoC Rust code taken to implement it. The slowdown in the mean and median time for $CPCT_{DM}^+$ suggests that configuration merging is particularly effective on files with complex or numerous errors.

6.3 The impact of skipping input

The number of error locations reported by panic mode is well over twice that of $CPCT^+$; even given $CPCT^+$'s higher failure rate relative to panic mode, this seemed hard to explain. We thus made an additional hypothesis:

H3 The greater the proportion of tokens that are skipped, the greater the number of error locations.

The intuition underlying this hypothesis is that, in general, the user's input is very close to being correct and that the more input that error recovery skips, the less likely it is to get back to a successful parse. We added the ability to record the proportion of tokens skipped



■ **Figure 13** A histogram of the number of files with 1–50 error locations for $CPCT^+$ and $CPCT^+_{rev}$. Note that we exclude: any files for which an error recovery algorithm did not find a single error location (either because the timeout was exceeded or no repair sequences could be found); and a handful of outliers which distort the full histogram found in Figure 18 in the Appendix. The x axis shows the number of error locations in a file and the y axis is a logarithmic scale for the number of files. Error bars represent 99% confidence intervals. As this histogram shows, the entire distribution is skewed slightly rightwards by $CPCT^+_{rev}$, showing that $CPCT^+_{rev}$ makes error recovery slightly worse in a number of files (rather than making error recovery in a few files a lot worse).

as the result of *delete* repairs during error recovery. The results in Figure 11 show a general correlation between the proportion of tokens skipped and the number of error locations (e.g. $CPCT^+$ skips very little of the user’s input; $CPCT^+_{rev}$ skips a little more; and panic mode skips an order of magnitude more). However, Corchuelo et al. does not obviously follow this pattern: relative to the other algorithms, its number of error locations does not correlate with the proportion of input skipped. This is mostly explained by its high mean time and high failure rate: Corchuelo et al. tends to fail on files with large numbers of error locations, underreporting the “true” number of error locations simply because it cannot make it all of the way through such files before the timeout. However, this outlier means that we consider Hypothesis H3 to be only weakly validated.

7 Using error recovery in practice

Although several members of the Fischer et al. family were implemented in parsing tools of the day, to the best of our knowledge none of those implementations have survived. Equally, we are not aware of any member of the Fischer et al. family which explains how error recovery should be used or, indeed, if it has any implications for users at all.

We are therefore forced to treat the following as an open question: can one sensibly use error recovery in the Fischer et al. family in practice? In particular, given that the most common way to use LR grammars is to execute semantic actions as each production is reduced, what should semantic actions do when parts of the input have been altered by error recovery? This latter question is important for real-world systems (e.g. compilers) which can still perform useful computations (e.g. running a type checker) in the face of syntax errors.

```

1 %start Expr
2 %%
3 Expr -> u64:
4     Factor "+" Expr { $1 + $3 }
5     | Factor { $1 }
6     ;
7
8 Factor -> u64:
9     Term "*" Factor { $1 * $3 }
10    | Term { $1 }
11    ;
12
13 Term -> u64:
14    "(" Expr ")" { $2 }
15    | "INT"
16    {
17        let n = $lexer.span_str($1.unwrap().span());
18        match s.parse::<u64>() {
19            Ok(val) => val as u64,
20            Err(_) => panic!("{0} cannot be represented as a u64", s)
21        }
22    }
23    ;

```

■ **Figure 14** A naive version of the calculator grammar with semantic actions on each production. The traditional Yacc `%union` declaration is unwieldy in Rust. Instead, *grmtools* allows each rule to have a Rust return type associated with it (between “->” and “:”): the actions of each production in that rule must return values of that type. `$n` variables reference the *n*th symbol in a production (where `$1` references the first symbol). If that symbol is a reference to a rule, a value of that rule’s Rust type will be stored in that variable. If that symbol is a token then the user’s input can be obtained by `$lexer.span_str($1.unwrap().span())` (line 17). Note that while this grammar’s semantic actions work as expected for inputs such as “2 + 3 * 4”, they will panic if too large a number is passed (line 20), or if an integer is inserted by error recovery. Figure 15 shows how to avoid both problems.

While different languages are likely to lend themselves to different solutions, in this section we show that *grmtools* allows sensible integration of error recovery in a Rust context. Readers who prefer to avoid Rust-specific details may wish to move immediately to Section 8.

7.1 A basic solution

Figure 14 shows a naive *grmtools* version of the grammar from Figure 2 that can evaluate numeric results as parsing occurs (i.e. given the input `2 + 3 * 4` it returns 14). This grammar should mostly be familiar to Yacc users: each production has a *semantic action* (i.e. Rust code that is executed when the production is reduced); and symbols in the production are available to the semantic action as pseudo-variables named `$n` (a production of *n* symbols has *n* pseudo-variables with the first symbol connected to `$1` and so on). A minor difference from traditional Yacc is that *grmtools* allows rules to specify a different return type, an approach shared with other modern parsers such as ANTLR [22].

A more significant difference relates to the `$n` pseudo-variables: if they reference a rule *R*, then their type is *R*’s return type; if they reference a token *T*, then their type is (slightly simplified) `Result<Lexeme, Lexeme>`. We will explain the reasons for this shortly, but at this stage it suffices to note that, unless a token was inserted by error recovery, we can extract tokens by calling `$1.unwrap()`, and obtain the actual string the user passed by using the globally available `$lexer.span_str` function.

```

1 %start Expr
2 %avoid_insert "INT"
3 %%
4 Expr -> Result<u64, Box<dyn Error>>:
5     Factor "+" Expr { Ok($1? + $3?) }
6     | Factor { $1 }
7     ;
8
9 Factor -> Result<u64, Box<dyn Error>>:
10    Term "*" Factor { Ok($1? * $3?) }
11    | Term { $1 }
12    ;
13
14 Term -> Result<u64, Box<dyn Error>>:
15    '(' Expr ')' { $2 }
16    | 'INT'
17    {
18        let t = $1.map_err(|_| "<evaluation aborted>");
19        let n = $lexer.span_str(t.span());
20        match s.parse::() {
21            Ok(val) => Ok(val as u64),
22            Err(_) => Err(Box::from(format!("{}", cannot be represented as a u64", s)))
23        }
24    }
25    ;

```

■ **Figure 15** A more sophisticated version of the grammar from Figure 14. Each rule now returns a **Result** type. If an integer is inserted by error recovery, the **Term** rule stops evaluation by percolating the **Err** value upwards using the “?” operator (which, if the **Result**-returning expression it is attached to evaluates to an **Err**, immediately returns that error; otherwise it unwraps the **Ok**); all other rules percolate such errors upwards similarly. As a convenience for the user, the contents of the “**Err**” value are changed from a lexeme to a string explaining why the calculator has not produced a value (line 18). Note that other token types are unaffected: if error recovery inserts a bracket, for example, evaluation of the expression continues.

7.2 Can semantic action execution continue in the face of error recovery?

In Yacc, semantic actions can assume that each symbol in the production has “normal” data attached to it (either a rule’s value or the string matching a token; Yacc’s error recovery is implicitly expected to maintain this guarantee) whereas, in our setting, inserted tokens have a type but no value. Given the input “(2 + 3)”, the inserted close bracket is not hugely important, and our calculator returns the value 5. However, given the input “2 +”, *CPCT*⁺ finds a single repair sequence [*Insert Int*]: what should a calculator do with an inserted integer? Our naive calculator simply **panics** (which is roughly equivalent to “raises an exception and then exits”) in such a situation (the **unwrap** in Figure 14 on line 17). However, there are two alternatives to this rather extreme outcome: the semantic action can assume a default value or stop further execution of semantic values while allowing parsing to continue. Determining which is the right action in the face of inserted tokens is inherently situation specific. We therefore need a pragmatic way for users to control what happens in such cases.

The approach we take is to allow users to easily differentiate normal vs. inserted tokens in a semantic action. Pseudo-variables that reference tokens have (slightly simplified) the Rust type **Result**<**Lexeme**, **Lexeme**>. Rust’s **Result** type⁸ is a sum type which represents success (**Ok**(...)) or error (**Err**(...)) conditions. We use the **Ok** case to represent “normal” tokens created from user input and the **Err** case to represent tokens inserted by error recovery. Since the **Result** type is widely used in Rust code, users can avail themselves of standard idioms.

⁸ Equivalents are found in several other languages: Haskell’s **Either**; O’Caml’s **result**; or Scala’s **Either**.

```
(a) . "ERRORTOKEN"
(b) ErrorRule -> ():
    "ERRORTOKEN" { }
    ;
(c) Parsing error at line 1 column 3. Repair sequences found:
    1: Insert +, Delete @
    2: Insert *, Delete @
    3: Delete @, Delete 3
    Result: 9
```

■ **Figure 16** A simple way of turning lexing errors into parsing errors in *grmtools*. First, we add a `ERRORTOKEN` token type, which matches otherwise invalid input, to the end of the Lex file (a). Second, we add a rule `ErrorRule` to the Yacc grammar referencing `ERRORTOKEN` (b). Note that `ErrorRule` must not be referenced by any other rule in the Yacc grammar. With those two steps complete, input with lexing errors such as “2 @ 3 + 4” invokes normal error recovery (c).

For example, we can then alter our calculator grammar to continue parsing, but stop executing meaningful semantic action code, when an inserted integer is encountered. We change grammar rules from returning type `T` to `Result<T, Box<dyn Error>>` (where `Box<dyn Error>` is roughly equivalent to “can return any type of error”). It is then, deliberately, fairly easy to use with the `Result<Lexeme, Lexeme>` type: for tokens whose value we absolutely require, we use Rust’s “?” operator (which passes `Ok` values through unchanged but returns `Err` values to the function caller) to percolate our unwillingness to continue evaluation upwards. While `Box<dyn Error>` is slightly verbose, it is a widely understood Rust idiom. Figure 15 shows that changing the grammar to make use of this idiom requires relatively little extra code.

7.3 Avoiding insert repairs when possible

Although we now have a reasonable mechanism for dealing with inserted tokens, there are cases where we can bypass them entirely. For example, consider the input “2 + + 3”, which has two repair sequences [*Delete +*], [*Insert Int*]: evaluation of the expression can continue with the former repair sequence, but not the latter. However, as presented thus far, these repair sequences are ranked equally and one non-deterministically selected.

We therefore added an optional declaration `%avoid_insert` to *grmtools* which allows users to specify those tokens which, if inserted by error recovery, are likely to prevent semantic actions from continuing execution. In practise, this is synonymous with those tokens whose values (and not just their types) are important. In the calculator grammar only the `INT` token satisfies this criteria, so we add `%avoid_insert "INT"` to the grammar. We then make a simple change to the repair sequence ranking of Section 5.3 such that the final list of repair sequences is sorted with inserts of such tokens at the bottom of the list. In our case, this means that we always select *Delete +* as the repair sequence to apply to the input “2 + + 3” (i.e. the *Insert Int* repair sequence is always presented as the second option).

7.4 Turning lexing errors into parsing errors

In most traditional parsing systems, lexing errors are distinct from parsing errors: only files which can be fully lexed can be parsed. This is confusing for users, who are often unaware of the distinction between these two phases. To avoid this, we lightly adapt the idea of *error tokens* from incremental parsing [31, p. 99]. In essence, any input which cannot be lexed is put into a token whose type is not referenced in the normal grammar. This guarantees that all possible input lexes without error and, when the parser encounters an error token, normal error recovery commences⁹.

⁹ Note that this is an opt-in feature: it was not enabled for the experiments in Section 6.

A basic, but effective, version of this requires no special support from *grmtools* (see Figure 16). First, we add a new token type to the lexer that matches each otherwise invalid input character. Second, since *grmtools* requires that all tokens defined in the lexer are referenced in the grammar, we add a dummy rule to the grammar that references the token (making sure not to reference this rule elsewhere in the grammar). These two steps are sufficient to ensure that users always see the same style of error messages, and the same style of error recovery, no matter whether they make a lexing or a parsing error.

8 Threats to validity

Although it might not be obvious at first, *CPCT*⁺ is non-deterministic, which can lead to different results from one run to the next. The root cause of this problem is that multiple repair sequences may have identical effects up to N_{try} tokens, but cause different effects after that value. By running each file through each error recovery multiple times and reporting confidence intervals, we are able to give a good – though inevitably imperfect – sense of the likely variance induced by this non-determinism.

Our implementation of Corchuelo et al. is a “best effort”. The description in the paper is incomplete in places and, to the best of our knowledge, the accompanying source code is no longer available. We thus may not have faithfully implemented the intended algorithm.

Blackbox contains an astonishingly large amount of source code but has two inherent limitations. First, it only contains Java source code. This means that our main experiment is limited to one grammar: it is possible that our techniques do not generalise beyond the Java grammar (though, as Appendix A suggests, our techniques do appear to work well on other grammars). Although [4, p. 109] suggests that different grammars make relatively little difference to the performance of such error recovery algorithms, we are not aware of an equivalent repository for other language’s source code. One solution is to mutate correct source files (e.g. randomly deleting tokens), thus obtaining incorrect inputs which we can later test: however, it is difficult to uncover and then emulate the numerous, sometimes surprising, ways that humans make syntax errors, particularly as some are language specific (though there is some early work in this area [7]). Second, Blackbox’s data comes largely from students, who are more likely than average to be somewhat novice programmers. It is clear that novice programmers make some different syntax errors – and, probably, make some syntax errors more often – relative to advanced programmers. For example, many of the files with the greatest number of syntax errors are caused by erroneous fragments repeated with variants (i.e. it is likely that the programmer wrote a line of code, copy and pasted it, edited it, and repeated that multiple times before deciding to test for syntactic validity). It is thus possible that a corpus consisting solely of programs from advanced programmers would lead to different results. We consider this a minor worry, partly because a good error recovery algorithm should aim to perform well with inputs from users of different experience levels.

Our corpus was parsed using a Java 1.7 grammar, but some members of the corpus were almost certainly written using Java 1.8 or later features. Many – though not all – post-1.7 Java features require a new keyword: such candidate source files would thus have failed our initial lexing test and not been included in our corpus. However, some Java 1.8 files will have made it through our checks. Arguably these are still a valid test of our error recovery algorithms. It is even likely that they may be a little more challenging on average, since they are likely to be further away from being valid syntax than files intended for Java 1.7.

9 Related work

Error recovery techniques are so numerous that there is no definitive reference or overview of them. However, [8] contains an overall historical analysis and [4] an excellent overview of much of the Fischer et al. family. Both must be supplemented with more recent works.

The biggest limitation of error recovery algorithms in the Fischer et al. family (including *CPCT⁺*) is that they find repairs at the point that an error is discovered, which may be later in the file than the cause of the error. Thus even when they successfully recover from an error, the repair sequence reported may be very different from the fix the user considers appropriate (note that this is distinct from the cascading error problem, which our ranking of repair sequences in Section 5.3 partly addresses). A common, frustrating, example of this is a missing “}” character in C/Java-like languages. Some approaches are able to backtrack from the source of the error in order to try and find more appropriate repairs. However, there are two challenges to this: first, the cost of maintaining the necessary state to backtrack slows down normal parsing (e.g. [6] only stores the relevant state at each line encountered to reduce this cost), whereas we add no overhead at all to normal parsing; second, the search-space is so hugely increased that it can be harder to find any repairs at all [8].

One approach to global error recovery is to use machine learning to train a system on syntactically correct programs [27]: when a syntax error is encountered, the resulting model is used to suggest appropriate global fixes. Although [27] also use data from Blackbox, their experimental methodology is both stricter – aiming to find exactly the same repair as the human user applied – and looser – they only consider errors which can be fixed by a single token, discarding 42% of the data [27, p. 8]) whereas we attempt to fix errors which span multiple tokens. It is thus difficult to directly compare our results to theirs. However, by the high bar they have set themselves, they are able to repair 52% of single-token errors.

As *CPCT_{rev}⁺* emphasises, choosing an inappropriate repair sequence during error recovery leads to cascading errors. The noncorrecting error recovery approach proposed by [26] explicitly addresses this weakness, eschewing repairs entirely. When a syntax error is discovered, noncorrecting error recovery attempts to discover all further syntax errors by checking whether the remaining input (after the point an error is detected) is a valid suffix in the language. This is achieved by creating a recogniser that can identify all valid suffixes in the language. Any errors identified in the suffix parse are guaranteed to be genuine syntax errors because they are uninfluenced by errors in the (discarded) prefix (though this does mean that some genuine syntax errors are missed that would not have been valid suffixes at that point in the user’s input had the original syntax error not been present). There seem to be two main reasons why noncorrecting error recovery has not been adopted. First, building an appropriate recogniser is surprisingly tricky and we are not currently aware of one that can handle the full class of LR grammars (though the full class of LL grammars has been tackled [30]), though we doubt that this problem is insoluble. Second, as soon as a syntax error is encountered, noncorrecting error recovery is unable to execute semantic actions, since it lacks the execution context they need.

Although one of our paper’s aims is to find the complete set of minimum cost repair sequences, it is unclear how best to present them to users, leading to questions such as: should they be simplified? should a subset be presented? and so on. Although rare, there are some surprising edge cases. For example, the (incorrect) Java 1.7 expression “`x = f("a"b);`” leads to 23,067 minimum cost repair sequences being found, due to the large number of Java keywords that are valid in several parts of this expression leading to a combinatorial explosion: even the most diligent user is unlikely to find such a volume of information valuable. In

a different vein, the success condition of “reached an accept” state is encountered rarely enough that we sometimes forgot that it could happen and were confused by an apparently unexplained difference in the repair sequences reported for the same syntax chunk when it was moved from the end to the middle of a file. There is a body of work which has tried to understand how best to structure compiler error messages (normally in the context of those learning to program). However, the results are hard to interpret: some studies find that more complex error messages are not useful [20], while others suggest they are [24]. It is unclear to us what the right approach might be, or how it could be applied in our context.

The approach of [17] is similar to Corchuelo et al., although the former cannot incorporate shift repairs. It tries harder than *CPCT*⁺ to prune out pointless search configurations [17, p. 12], such as cycles in the parsing stack, although this leads to some minimum cost repairs being skipped [2]. A number of interlocking, sophisticated pruning mechanisms which build on this are described in [4]. These are significantly more complex than our merging of compatible configurations: since this gives us acceptable performance in practise, we have not investigated other pruning mechanisms.

The most radical member of the Fischer et al. family is that of [15]¹⁰. This generates repair sequences in the vein of Corchuelo et al. using the A* algorithm and a precomputed distance table. [15] works exclusively on the stategraph, assuming that it is unambiguous. However, Yacc systems allow ambiguous stategraphs and provide a means for resolving those ambiguities when creating the statetable. Many real-world grammars (e.g. Lua, PHP) make use of ambiguity resolution. In an earlier online draft, we created *MF*, a statetable-based algorithm which extends *CPCT*⁺ with ideas from [15] at the cost of significant additional complexity. With the benefit of hindsight, we do not consider *MF*'s relatively small benefits (e.g. reducing the failure rate by approximately an additional 0.5%) to be worth that additional complexity.

CPCT⁺ takes only the grammar and token types into account. However, it is possible to use additional information, such as nesting (e.g. taking into account curly brackets) and indentation when recovering from errors. This has two aims: reducing the size of the search space (i.e. speeding up error recovery); and making it more likely that the repairs reported matched the user's intentions. The most sophisticated approach in this vein we are aware of is that of [6]. At its core, this approach uses GLR parsing: after a grammar is suitably annotated by the user, it is then transformed into a “permissive” grammar which can parse likely erroneous inputs; strings which match the permissive parts of the grammar can then be transformed into a non-permissive counterpart. In all practical cases, the transformed grammar will be ambiguous, hence the need for generalised parsing. Our use of parent-pointer trees in configuration merging gives that part of our algorithm a similar feel to GLR parsing (even though we do not generate ambiguous strings). However, there are major differences: LR parsers are much simpler than GLR parsers; and the Fischer et al. family of algorithms do not require manually annotating, or increasing the size of, the grammar.

A different approach to error recovery is that taken by [23]: rather than try and recover from errors directly, it reports in natural language how the user's input caused the parser to reach an error state (e.g. “I read an open bracket followed by an expression, so I was expecting a close bracket here”), and possible routes out of the error (e.g. “A function or variable declaration is valid here”). This involves significant manual work, as every parser

¹⁰In an earlier online draft of this paper we stated that this algorithm has a fundamental flaw. We now believe this was due to us incorrectly assuming that the “delete” optimisation of Corchuelo et al. applied to [15]. We apologise to the authors for this mistake.

state (1148 in the Java grammar we use) in which an error can occur needs to be manually marked up, though the approach has various techniques to lessen the problem of maintaining messages as a grammar evolves.

Many compilers and editors have hand-written parsers with hand-written error recovery. Though generally ad-hoc in their approach, it is possible, with sufficient effort, to make them perform well. However, this comes at a cost. For example, the hand-written error recovery routines in the Eclipse IDE are approximately 5KLoC and are solely for use with Java code: *CPCT*⁺ is approximately 500LoC and can be applied to any LR grammar.

Although error recovery approaches have, historically, been mostly LR based, there are several non-LR approaches. A full overview is impractical, though a few pointers are useful. When LL parsers encounter an error, they generally skip input until a token in the follow set is encountered (an early example is [29]). Although this outperforms the simple panic mode of Section 3, it will, in general, clearly skip more input than *CPCT*⁺, which is undesirable. LL parsers do, however, make it somewhat easier to express grammar-specific error recovery rules. The most advanced LL approach that we are aware of is IntelliJ's Grammar-Kit, which allows users to annotate their grammars for error recovery. Perhaps the most interesting annotation is that certain rules can be considered as fully matched even if only a prefix is matched (e.g. a partially completed function is parsed as if it was complete). It might be possible to add similar ideas to a successor of *CPCT*⁺, though this is more awkward to express in an LR approach. Error recovery for PEG grammars is much more challenging, because the non-LL parts of the grammar mean that there is not always a clearly defined point at which an error is determined to have occurred. PEG error recovery has thus traditionally required extensive manual annotations in order to achieve good quality recovery. [18] tackles this problem by automatically adding many (though not necessarily all) of the annotations needed for good PEG error recovery. However, deciding when to add, and when not to add, annotations is a difficult task and the two algorithms presented have different trade-offs: the *Standard* algorithm adds more annotations, leading to better quality error recovery, but can change the input language accepted; the *Unique* algorithm adds fewer annotations, leading to poorer quality error recovery, but does not affect the language accepted. The quality of error recovery of the Unique algorithm, in particular, is heavily dependent on the input grammar: it works well on some (e.g. Pascal) but less well on others (e.g. Java). In cases where it performs less well, it can lead to parsers which skip large portions (sometimes the remainder) of the input.

While the field of programming languages has largely forgotten the approach of [1], there are a number of successor works (e.g. [25]). These improve the time complexity, though none that we are aware of address the issue of how to present to the user what has been done.

We are not aware of any error recovery algorithms that are formally verified. Indeed, as shown in this paper, some have serious flaws. We are only aware of two works which have begun to consider what correctness for such algorithms might mean: [32] provides a brief philosophical justification of the need and [12] provides an outline of an approach. Until such time as someone verifies a full error recovery algorithm, it is difficult to estimate the effort involved, or what issues may be uncovered.

10 Conclusions

In this paper we have shown that error recovery algorithms in the Fischer et al. family can run fast enough to be usable in the real world. Extending such algorithms to produce the complete set of minimum cost repair sequences allows parsers to provide better feedback to users, as well as significantly reducing the cascading error problem. The *CPCT*⁺ algorithm is simple to implement (less than 500LoC in our Rust system) and still has good performance.

Looking to the future, we, perhaps immodestly, suggest that $CPCT^+$ might be “good enough” to serve as a common representative of the Fischer et al. family. However, we do not think that it is the perfect solution. We suspect that, in the future, multi-phase solutions will be developed. For example, one may use noncorrecting error recovery (e.g. [26]) to identify syntax errors, and then use a combination of machine-learning (e.g. [27]) and $CPCT^+$ to discover those repair sequences that do not lead to additional error locations being encountered.

References

- 1 Alfred Aho and Thomas G. Peterson. A minimum distance error-correcting parser for context-free languages. *J. Comput.*, 1:305–312, December 1972.
- 2 Eberhard Bertsch and Mark-Jan Nederhof. On failure of the pruning technique in “error repair in shift-reduce parsers”. *TOPLAS*, 21(1):1–10, January 1999.
- 3 Neil C. C. Brown, Michael Kölling, Davin McCall, and Ian Utting. Blackbox: A large scale repository of novice programmers activity. In *SIGCSE*, March 2014.
- 4 Carl Cerecke. *Locally least-cost error repair in LR parsers*. PhD thesis, University of Canterbury, June 2003.
- 5 Rafael Corchuelo, José Antonio Pérez, Antonio Ruiz-Cortés, and Miguel Toro. Repairing syntax errors in LR parsers. *TOPLAS*, 24:698–710, November 2002.
- 6 Maartje de Jonge, Lennart C. L. Kats, Eelco Visser, and Emma Söderberg. Natural and flexible error recovery for generated modular language environments. *TOPLAS*, 34(4):15:1–15:50, December 2012.
- 7 Maartje de Jonge and Eelco Visser. Automated evaluation of syntax error recovery. In *ASE*, pages 322–325, September 2012.
- 8 Pierpaolo Degano and Corrado Priami. Comparison of syntactic error handling in LR parsers. *SPE*, 25(6):657–679, June 1995.
- 9 Joel E. Denny and Brian A. Malloy. The IELR(1) algorithm for generating minimal LR(1) parser tables for non-LR(1) grammars with conflict resolution. *SCICO*, 75(11):943–979, November 2010.
- 10 Bradley Efron. Bootstrap methods: Another look at the jackknife. *Ann. Statist.*, 7(1):1–26, January 1979.
- 11 C. N. Fischer, B. A. Dion, and J. Mauney. A locally least-cost LR-error corrector. Technical Report 363, University of Wisconsin, August 1979.
- 12 Carlos Gómez-Rodríguez, Miguel A. Alonso, and Manuel Vilares. Error-repair parsing schemata. *TCS*, 411(7):1121–1139, February 2010.
- 13 Allen I. Holub. *Compiler Design in C*. Prentice-Hall, 1990.
- 14 S. C. Johnson. YACC: Yet Another Compiler-Compiler. Technical Report Comp. Sci. 32, Bell Labs, July 1975.
- 15 Ik-Soon Kim and Kwangkeun Yi. LR error repair using the A* algorithm. *Acta Inf.*, 47:179–207, May 2010.
- 16 Donald Knuth. On the translation of languages from left to right. *Information and Control*, 8(6):607–639, December 1965.
- 17 Bruce J. McKenzie, Corey Yeatman, and Lorraine de Vere. Error repair in shift-reduce parsers. *TOPLAS*, 17(4):672–689, July 1995.
- 18 Sérgio Medeiros, Gilney de Azevedo Alvez Junior, and Fabio Mascarenhas. Syntax error recovery in parsing expression grammars. *CoRR*, May 2019. [arXiv:1905.02145](https://arxiv.org/abs/1905.02145).
- 19 Sérgio Medeiros and Fabio Mascarenhas. Syntax error recovery in parsing expression grammars. In *SAC*, pages 1195–1202, April 2018.
- 20 Marie-Hélène Nienaltowski, Michela Pedroni, and Bertrand Meyer. Compiler error messages: What can help novices? In *SIGCSE*, pages 168–172, March 2008.

- 21 David Pager. A practical general method for constructing LR(k) parsers. *Acta Inf.*, 7(3):249–268, September 1977.
- 22 Terence Parr. *The definitive ANTLR 4 reference*. Pragmatic Bookshelf, January 2013.
- 23 François Pottier. Reachability and error diagnosis in LR(1) parsers. In *CC*, pages 88–98, March 2016.
- 24 James Prather, Raymond Pettit, Kayla Holcomb McMurry, Alani Peters, John Homer, Nevan Simone, and Maxine Cohen. On novices' interaction with compiler error messages: a human factors approach. In *ICER*, pages 74–82, August 2017.
- 25 Sanguthevar Rajasekaran and Marius Nicolae. An error correcting parser for context free grammars that takes less than cubic time. In *LATA*, February 2016.
- 26 Helmut Richter. Noncorrecting syntax error recovery. *TOPLAS*, 7(3):478–489, July 1985.
- 27 Eddie Antonio Santos, Joshua Charles Campbell, Dhvani Patel, Abram Hindle, and José Nelson Amaral. Syntax and sensibility: Using language models to detect and correct syntax errors. In *SANER*, pages 1–11, March 2018.
- 28 Masaru Tomita. An efficient augmented-context-free parsing algorithm. *Comput. Linguist.*, 13(1-2):31–46, January 1987.
- 29 D. A. Turner. Error diagnosis and recovery in one pass compilers. *IPL*, 6(4):113–115, August 1977.
- 30 Arthur van Deudekom, Dick Grune, and Peter Kooiman. Initial experience with noncorrecting syntax error handling. Technical Report IR-339, Vrije Universiteit, Amsterdam, November 1993.
- 31 Tim A. Wagner. *Practical Algorithms for Incremental Software Development Environments*. PhD thesis, University of California, Berkeley, March 1998.
- 32 Vadim Zaytsev. Formal foundations for semi-parsing. In *SMR*, pages 313–317, February 2014.

Appendix (not peer-reviewed)

A Curated examples

In this section we show several examples of error recovery using $CPCT^+$ in different languages, to give some idea of what error recovery looks like in practise, and to emphasise that the algorithms in this paper are grammar neutral. All of these examples use the output from the *nimbleparse* tool that is part of *grmtools*.

A.1 Java 7

Example 1 input:

```

1 class C {
2   int x y;
3 }
```

Example 1 output:

```

Parsing error at line 2 column 9. Repair sequences found:
  1: Insert ,
  2: Delete y
  3: Insert =
```

Example 2 input:

```

1 class C {
2   void f() {
3     if true {
4   }
5 }
```

Example 2 output:

```
Parsing error at line 3 column 8. Repair sequences found:
1: Insert (, Shift true, Insert )
Parsing error at line 5 column 2. Repair sequences found:
1: Insert }
```

Example 3 (taken from [6, p. 10]) input:

```
1 class C {
2   void f() {
3     if (temp.greaterThan(MAX) // missing )
4       fridge.startCooling();
5   }
6 }
```

Example 3 output:

```
Parsing error at line 4 column 7. Repair sequences found:
1: Insert )
```

Example 4 (taken from [6, p. 16]) input:

```
1 class C {
2   void methodX() {
3     if (true)
4       foo();
5   }
6   int i = 0;
7   while (i < 8)
8     i=bar(i);
9   }
10 }
11 }
```

Example 4 output:

```
Parsing error at line 7 column 5. Repair sequences found:
1: Insert {
Parsing error at line 11 column 1. Repair sequences found:
1: Delete }
```

Example 5 (taken from [19, p. 2]):

```
1 public class Example {
2   public static void main(String[] args) {
3     int n = 5;
4     int f = 1;
5     while(0 < n) {
6       f = f * n;
7       n = n - 1
8     };
9     System.out.println(f);
10  }
11 }
```

Example 5:

```
Parsing error at line 8 column 5. Repair sequences found:
1: Insert ;
2: Delete }
```

Example 6:

```
1 class C {
2   void f() {
3     x((((((
4   }
5 }
```

6:30 Don't Panic! Better, Fewer, Syntax Errors for LR Parsers

Example 6 output, showing the timeout being exceeded and error recovery unable to complete:

```
Parsing error at line 4 column 3. No repair sequences found.
```

A.2 Lua 5.3

Example 1 input:

```
1 print("Hello World")
```

Example 1 output:

```
Parsing error at line 1 column 20. Repair sequences found:  
1: Insert )
```

Example 2 input. Note that “=” in Lua is the assignment operator, which is not valid in conditionals; and that if/then/else blocks must be terminated by “end”.

```
1 function fact (n)  
2   if n = 0 then  
3     return 1  
4   else  
5     return n * fact(n-1)  
6 end
```

Example 2 output:

```
Parsing error at line 2 column 8. Repair sequences found:  
1: Insert .., Delete =  
2: Insert //, Delete =  
3: Insert -, Delete =  
4: Insert %, Delete =  
5: Insert *, Delete =  
6: Insert >, Delete =  
7: Insert <<, Delete =  
8: Insert ^, Delete =  
9: Insert <=, Delete =  
10: Insert +, Delete =  
11: Insert /, Delete =  
12: Insert or, Delete =  
13: Insert <, Delete =  
14: Delete =, Delete 0  
15: Insert and, Delete =  
16: Insert ==, Delete =  
17: Insert ~=, Delete =  
18: Insert |, Delete =  
19: Insert ~, Delete =  
20: Insert >=, Delete =  
21: Insert &, Delete =  
22: Insert >>, Delete =  
Parsing error at line 6 column 4. Repair sequences found:  
1: Insert end
```

Examples 3 and 4 (both derived from the Lua 5.3 reference manual) show that *CPCT*⁺ naturally deals with an inherent ambiguity in Lua's Yacc grammar involving function calls and assignments (which, following the Lua specification, is resolved by Yacc in favour of function calls). Example 3 shows the “unambiguous” case (i.e. if Lua forced users to use “;” everywhere, the grammar would have no ambiguities):

```
1 a = b + c;  
2 (print or io.write)'done')
```

Example 3 output:

```
Parsing error at line 2 column 26. Repair sequences found:  
1: Delete )  
2: Insert (
```


Example 4 shows what happens in the “ambiguous” case (which Lua’s grammar resolves in favour of viewing the code below as a function call to c):

```
1 a = b + c
2 (print or io.write)'done')
```

Example 4 output:

```
Parsing error at line 2 column 26. Repair sequences found:
1: Delete )
```

Example 5 (taken from [19, p. 7]):

```
1 if then print("that") end
```

Example 5 output:

```
Parsing error at line 1 column 4. Repair sequences found:
1: Insert <Name>
2: Insert <Numeral>
3: Insert true
4: Insert "<String>"
5: Insert "[[<String>]=]"
6: Insert nil
7: Insert false
8: Insert ...
```

A.3 PHP 7.3

Example 1 input:

```
1 function n() {
2     $x = 1
3 }
```

Example 1 output:

```
Parsing error at line 3 column 1. Repair sequences found:
1: Insert ;
```

Example 2 input:

```
1 $a = array("foo", "bar");
2 $a[0;
```

Example 2 output:

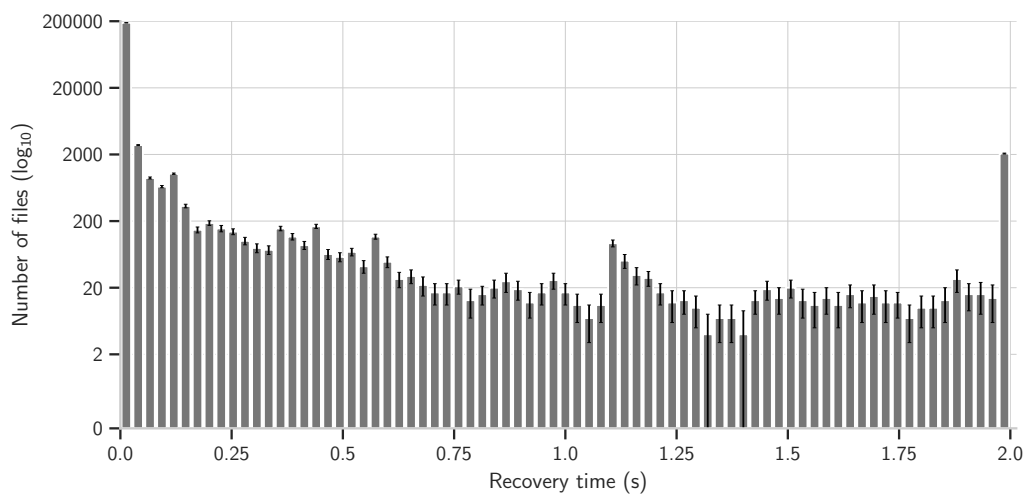
```
Parsing error at line 2 column 5. Repair sequences found:
1: Insert ]
```

Example 3 input:

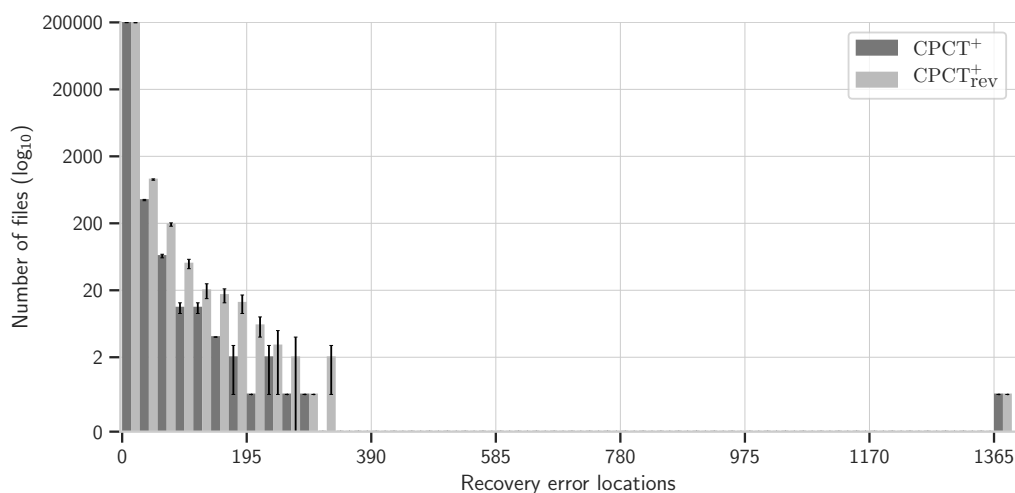
```
1 class X {
2     function a($x) {
3         if $x {
4         }
5 }
```

Example 3 output:

```
Parsing error at line 3 column 12. Repair sequences found:
1: Insert (, Shift $x, Insert )
Parsing error at line 5 column 2. Repair sequences found:
1: Insert }
```



■ **Figure 17** A histogram showing the effect of increasing the timeout from 0.5s to 2s (see Figure 12 for the histogram showing a timeout of 0.5s). Increasing the timeout from 0.5s to 2s lowers the failure rate from $1.63\% \pm 0.017\%$ to $1.02\% \pm 0.016\%$, with a slowly decreasing number of files succeeding as the timeout increases. Although we used a timeout of 0.5s on the basis that we felt most users would tolerate such a delay, others may wish to pick a shorter or longer timeout depending on their perception of their users tolerance of delay vs. tolerance of failed error correction.



■ **Figure 18** The full histogram of the number of error locations. The small number of outliers obscures the main bulk of the data – see Figure 13 for the truncated version.

K-LLVM: A Relatively Complete Semantics of LLVM IR

Liyi Li

Department of Computer Science, University of Illinois at Urbana-Champaign, Urbana, IL, USA
liyili2@illinois.edu

Elsa L. Gunter

Department of Computer Science, University of Illinois at Urbana-Champaign, Urbana, IL, USA
egunter@illinois.edu

Abstract

LLVM [21] is designed for the compile-time, link-time and run-time optimization of programs written in various programming languages. The language supported by LLVM targeted by modern compilers is LLVM IR [29]. In this paper we define **K-LLVM**, a reference semantics for LLVM IR. To the best of our knowledge, **K-LLVM** is the most complete formal LLVM IR semantics to date, including all LLVM IR instructions, intrinsic functions in the LLVM documentation and Standard-C library functions that are necessary to execute many LLVM IR programs. Additionally, **K-LLVM** formulates an abstract machine that executes all LLVM IR instructions. The machine allows to describe our formal semantics in terms of simulating a conceptual virtual machine that runs LLVM IR programs, including non-deterministic programs. Even though the **K-LLVM** memory model in this paper is assumed to be a sequentially consistent memory model and does not include all LLVM concurrency memory behaviors, the design of **K-LLVM**'s data layout allows the **K-LLVM** abstract machine to execute some LLVM IR programs that previous semantics did not cover, such as the full range of LLVM IR behaviors for the interaction among LLVM IR casting, pointer arithmetic, memory operations and some memory flags (e.g. `readonly`) of function headers. Additionally, the memory model is modularized in a manner that supports investigating other memory models. To validate **K-LLVM**, we have implemented it in \mathbb{K} [41], which generated an interpreter for LLVM IR. Using this, we ran tests including 1,385 unit test programs and around 3,000 concrete LLVM IR programs, and **K-LLVM** passed all of them.

2012 ACM Subject Classification Theory of computation → Operational semantics

Keywords and phrases LLVM, formal semantics, K framework, memory model, abstract machine

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2020.7

Funding This material is based upon work supported in part by NSF Grant CCF 13-18191. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the NSF.

1 Introduction

The Low Level Virtual Machine (LLVM) is designed for the compile-time, link-time and run-time optimizations of programs written in unspecified programming languages. An LLVM-based compiler, such as Clang, relies on a translation from a high-level source language to an intermediate representation (LLVM IR) that hides details about the specific target execution platform and acts as an interface for LLVM. Then, users are able to use the LLVM tools to perform program optimizations, transformations, and static analyses based on LLVM IR, which can also be translated into target architectures such as x86, PowerPC, and ARM. Hence, LLVM IR acts as a “central station” for translating high-level languages to target architectures, with a fixed set of language syntax, instructions, library functions, and a memory model [29].



© Liyi Li and Elsa L. Gunter;
licensed under Creative Commons License CC-BY
34th European Conference on Object-Oriented Programming (ECOOP 2020).

Editors: Robert Hirschfeld and Tobias Pape; Article No. 7; pp. 7:1–7:30

Leibniz International Proceedings in Informatics



LIPICs Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

When using LLVM IR in Clang, the correctness of executing programs is a big concern. Previous work [44, 30] has identified more than 200 LLVM compiler bugs. To verify Clang, we first need to know what the correct behavior of LLVM IR is. However, there are several issues about the currently existing language specifications related to LLVM IR. This paper provides an overview of a relatively complete semantics addressing these issues. One challenge to giving a complete semantics is its sheer size. To the best of our knowledge, VeLLVM [45] is the only notable and published attempt to give LLVM IR a formal semantics, and it only provides a limited subset of LLVM IR features, which does not include the LLVM library functions, a multi-threaded memory model, or the standard-C library functions. A second challenge is finding the right balance between mathematical abstractions and real world concrete details about the LLVM IR semantics. Most of the previous work [45, 18, 17] has utilized mathematical abstractions for the LLVM IR semantics so that theorems could be proved in an elegant and simple way. However, LLVM IR is not a high enough level of language that such abstractions can reflect the full semantics with total precision. Its semantics contains a lot of detailed information that a real implementable semantics needs to explain. For example, even though VeLLVM allows memory alignments, it does not allow memory operations to have alignment information. In LLVM IR, if the alignment value for a memory operation is not set properly, the behavior can be undefined. The lack of such information means that VeLLVM lacks the definition of important features of LLVM IR. Filling in all these details is challenging but important for defining the whole LLVM IR semantics. Third, even if one has the details in their semantics, we still need a good way to combine them together to form a unified framework with simplicity and modularity. For example, the C semantics in \mathbb{K} [11] has considered many details of the memory layouts necessary for executing C programs. However, its execution and memory models are so basic that any extension of the semantics requires a major change in them, such as the extension of atomic memory operations. In this paper, a rigorous executable specification is formalized for the LLVM IR language to overcome these problems. Our **K-LLVM** semantics defines almost all of the features in LLVM IR that are listed in the LLVM IR documentation (see Limitations at the end of this section), which has more than 219 pages. **K-LLVM** also offers a unified framework as an abstract machine that executes LLVM IR programs. The framework allows us to cover all corner case semantics of LLVM IR operations. The full details of our semantics can be found in the **K-LLVM** implementation [26]. This paper highlights an interesting portion of **K-LLVM** to show how one can possibly find a balance between abstractions and real world programming to provide a better, clearer, and more useful language semantics. First, we introduce some benefits, features, and a limitation of **K-LLVM**.

The Most Complete LLVM IR Semantics. **K-LLVM** is the most complete LLVM IR semantics to date, and provides a reference for people to use when exploring LLVM IR behaviors, including threading behaviors. The semantics is complete relative to a byte-wise, sequentially consistent memory model. **K-LLVM** defines corner cases for all LLVM IR operations, some of which have not been defined by previous work.

A Unified and Rigorously Mathematical Framework. We provide a unified and rigorously mathematical framework where people can observe the semantic behaviors in a single interface and also prove properties of compilers, with a focus on LLVM IR and LLVM IR compilers. Transforming programs from a high-level language to a low-level machine code requires a lot of phases, each of which might cause correctness concerns. For example, the infamous out-of-

thin-air problems can arise at every level of intermediate AST as a result of a transformation or compiler optimization. They can even appear when some old processors try to execute certain programs [35]. **K-LLVM** provides a way for users to reason about the behaviors of these translations based on the rigorous executable semantics of LLVM IR.

A Conceptual Device and a Virtual Machine. **K-LLVM** is implemented as a virtual machine that runs LLVM IR codes, that are interpretable by users. Instead of having to understand axiomatized memory events, they deal with central processors, threads, memory caches, etc. **K-LLVM** accomplishes this by providing an abstract machine that combines its runtime system, executions and memory models (in byte-wise sequential consistency). It implements the executable LLVM IR semantics for version 6.0.0. The abstract machine is also scalable. With simple changes to the current **K-LLVM**, the machine can allow the LLVM IR instructions to be executed out of order, handle speculative executions, and simulate a real-world memory environment that allows for features such as memory caches.

Detailed LLVM IR Low-level Structure. LLVM IR is a low enough language that one cannot define the semantics without explicitly incorporating aspects of the underlying architecture. It is important to deal with low-level data values like integers, floats, and pointers in a more detailed format based on bits and bytes, instead of pure mathematical concepts (see Section 3.2).

Parametric Behavior. **K-LLVM** has been implemented in a direct and transparent manner in \mathbb{K} , resulting in an interpreter for LLVM IR. **K-LLVM** is parameterized by important information needed for implementing defined behaviors. Users can configure the parameters of the semantics based on specific architectures or compilers, and then proceed to see executable behaviors formally in the implementation in \mathbb{K} .

Undefined Behavior. We classify three different types of undefinedness in LLVM IR. The first one is `undef`, which represents an unspecified value for a program position; the program should proceed no matter what the value is. In some cases, `undef` also means that the program has ill-defined behavior, such as representing a race in the memory. There are two ways to deal with `undef` in **K-LLVM**: *krun* can be used to execute a program with `undef` and get a fixed deterministic behavior by assuming one path, or *ksearch* can be used to search for all different behaviors by executing the program non-deterministically. Sometimes, the non-deterministic search space caused by `undef` values in LLVM IR is too large. In such cases, the symbolic execution engine in *ksearch* with the \mathbb{K} equivalence checker can be used to determine if two programs return the same results. Additional discussion can be found in Section 5. The second kind of undefinedness is an undefined behavior represented by a poison value, because LLVM IR does not have a defined symbol for it. Its meaning is similar to `undef`, but it has certain undefined behaviors associated with it. **K-LLVM** will carry the poison value and continue computation until a non-deterministic point is reached, then give an error message saying that there is a poison value in the program, and stop the continuation of the computation. If no non-deterministic point is found, **K-LLVM** can finish the computation successfully. The third kind results from underspecification in the LLVM IR documentation. We named this as **unspecified behaviors** in this paper. When facing the third kind, **K-LLVM** immediately labels the computation an error state, saying there is an unspecified behavior in the system. More information can be found in Section 5.

Independent of \mathbb{K} . The implementation in \mathbb{K} gives **K-LLVM** the power to have an interpreter automatically, and have tools for state space searching and symbolic executions. Essentially, \mathbb{K} [42] is an executable semantic framework based a rewriting logic [34]. Once a language semantics is defined in \mathbb{K} , it automatically turns it into a logical form by turning each semantic rule into an axiomatic rule with pre and post-conditions; thus, it creates an axiom set for the language. Additionally, there are many tools available in \mathbb{K} . For example, *kompile* can be used to see if the semantics has static type problems and to generate an interpreter, so that *krun* can be used with the interpreter to test their semantics by actual concrete programs. *ksearch* allows searches of traces of multi-threaded programs based on the interpreter. The symbolic engine in *ksearch* and the program equivalence checker in \mathbb{K} can allow for two sets of traces to be compared by symbolically executing two different multi-threaded programs and seeing if the two sets produce the same output. Even though we have defined **K-LLVM** in \mathbb{K} , the semantics is independent of its implementations in \mathbb{K} . In fact, we have defined the **K-LLVM** abstract machine in Isabelle [39] for manually proving theorems about **K-LLVM**. Additional discussion is presented in Section 5.

Limitations. In this paper, the **K-LLVM** memory model is based on byte-wise sequential consistency. LLVM IR specifies a range of behaviors for memory operations with different orderings and for `volatile` memory accesses, while **K-LLVM** does not support the full range. In **K-LLVM**, every memory location is mapped to a single byte datum; there is only one memory cache to deal with all memory operation requests from the different threads. Single thread instruction execution is in the program order. Based on this model with the **K-LLVM** abstract machine, we provide an observation in Section 4.4. We implemented the full LLVM IR concurrency model (based on the `glibc` and `pthread_create` libraries) in \mathbb{K} with all of the memory ordering behaviors of the atomic memory operations, but we have not yet finished proving the properties of that model; so it will be a future extension of **K-LLVM**. The work is described in the technical report [26].

2 Related Work

The **K-LLVM** semantics builds on top of the LLVM semantics in \mathbb{K} by Ellison and Lazar [10]. Our semantics directly extends their work to support missing features, including a more precise memory model and concurrency. Here we provide a description of other projects related to the definition of the LLVM IR semantics, and also review large language specifications related to our design.

Other LLVM IR Semantics. Besides **K-LLVM**, the other formal executable semantics for LLVM-IR are VeLLVM [24] and the previous LLVM semantics in \mathbb{K} [10]. VeLLVM was the first project to define a relatively complete specification for the core of LLVM IR. It is defined in the theorem prover Coq [2] and covers a core set of LLVM IR instructions. VeLLVM formalizes a mechanized semantics for LLVM IR, its type system, and the properties of its SSA form. It also has an interpreter extracted from Coq that ran 145 test programs and passed 134 of them. The memory model of VeLLVM is based on CompCert [3, 25] with newly developed features that are specifically designed by the VeLLVM team for capturing the memory data layout features in LLVM IR. Their model associates metadata to memory byte data fields, so that an execution of a LLVM IR program can utilize the metadata. This feature is similar to the memory data layout in **K-LLVM** (see Sec. 4.3). With VeLLVM, users can prove properties about translations defined in LLVM IR. Several papers, such as [24, 17], have been published about compiler correctness, memory models, and verification of compiler schemes using VeLLVM.

There are two levels of difference between **K-LLVM** and VeLLVM. The first level is the quantitative level. **K-LLVM** defines semantics for a larger set of LLVM IR operations, LLVM IR operation features, and library functions than what VeLLVM define. This is partly because the original VeLLVM semantics was based on LLVM 2.0, while **K-LLVM** is built on top of LLVM 6.0.0. There are also features that VeLLVM claimed not to define but **K-LLVM** covers, such as the different calling conventions. **K-LLVM** allows users to set up different calling conventions for the abstract machine that executes **K-LLVM** programs, and these different conventions show different execution behaviors in function call semantics. The second level is the qualitative level. **K-LLVM** formalizes the LLVM semantics using a more directly operational method, reflecting the possible implementation of semantics in a virtual computer, while VeLLVM focuses on the formalization of LLVM semantics as a mathematical object. The virtual computer upon which the **K-LLVM** semantics is built is split into common conceptual computer components, and the **K-LLVM** semantics investigates the interactions between different LLVM IR operations and these different components.

There are three main differences between **K-LLVM** and VeLLVM on the qualitative level. The first difference is that the abstract machine underlying **K-LLVM** supports the memory object model under the multi-threaded environment, while VeLLVM is single-threaded, which is all the LLVM documentation [29] specifies for the LLVM IR. The second difference is the handling of the stack and heap memories. VeLLVM implements the stack and heap using the same allocation semantics, allocating blocks in main memory. On the other hand, **K-LLVM** views them as different components, with each thread having its own size-limited stack accessed by `alloca`, but with one shared heap accessed by `malloc`. The different ways that each of VeLLVM and **K-LLVM** provides semantics of memory usage in LLVM IR show one of the key philosophical differences between the approaches of the two systems. It's true that the LLVM documentation does not mention the difference between stack and heap, therefore it is correct for VeLLVM to use the same semantics for stack and heap allocations. However, LLVM is a low-level language that contains certain machine level features. A major design feature of **K-LLVM** is to formalize an LLVM semantics based on a general underlying machine structure, which captures many real-world machine features. These features include being able to support `glibc` libraries, and the ability to express the LLVM IR multi-threaded semantics based on the interactions among different components in the machine. For these reasons, **K-LLVM** needs a more complex and low-level specification for `malloc` and `alloca` operations.

The third difference is formalization of memory location objects (layout structures). Both VeLLVM and **K-LLVM** have special byte data structures including metadata to keep track of pointer provenance information in a memory location, but VeLLVM has two different byte data structures, one for byte data that are originally stored as basic data such as integers or floats, and one for byte data that are originally stored as pointers. In addition, VeLLVM does not have special structures for basic data values in the registers to carry these metadata. This allows **K-LLVM** to give semantics to more valid LLVM IR programs. For example, assume that we execute the following LLVM program piece in VeLLVM and **K-LLVM**.

```

1  %r1 = call i8* @malloc (i64 4)
2  %r2 = call i8* @malloc (i64 4)
3  %r3 = bitcast i8* %r1 to i64*
4  %r4 = ptrtoint i8* %r2 to i64
5  store i64 %r4 , i64* %r3
6  %r5 = bitcast i64* %r3 to i64**
7  %r6 = load i64*, i64** %r5
8  store i64 1, i64* %r6
...

```

The execution of the above program piece in VeLLVM gets stuck at the line 8 of the program, while K-LLVM successfully finishes execution of it, as prescribed by the LLVM documentation. In VeLLVM, once a pointer is cast to an integer and then stored at a memory location, the meta provenance information for the pointer is lost. If the pointer is loaded back and used as a pointer for communicating with the memory, the behavior is forbidden in VeLLVM. This loss of information was also the motivation for the VeLLVM researchers to develop a C memory model that supports integer-pointer casts [17]. In **K-LLVM**, we keep track of the provenance information for the lifetime of the pointer no matter what the pointer becomes. The details are in Sec. 4.4.

Other Work Related to the LLVM IR Semantics. There are other pieces of work that are not meant to directly define the LLVM IR semantics but influence **K-LLVM**. First, Lee et al. [24] investigated the LLVM IR undefined behaviors with no concrete semantics for all undefined behaviors. Kang et al. [17] provided a model in C to support the `inttoptr/ptrtoint` casting operations. Their work enlightened **K-LLVM**. However, their definition focused on the aspect of a memory model, leaving the execution of programs as a black box. Thus, their casting operation semantics does not work with the real LLVM IR semantics. Ellison and Rosu [11] defined the full C semantics with a simplified version of CompCert’s model. Chakraborty and Vafeiadis [7] provided a concurrent abstracted memory model for LLVM IR that focused on an abstraction of the concurrent LLVM IR memory behaviors. Lee et al. proposed a novel LLVM memory model including a data layout and memory pointer provenance model [23]. They claimed to provide a better LLVM memory model that was sound and performed better. However, their model targeted a very small set of LLVM IR memory related instructions, and their abstract machine was simple. It is unclear how their model can be extended to include the behaviors of other LLVM IR instructions, especially the side-effects caused by interactions between different instructions, such as the additional behaviors caused by having the `readonly` flag or the thread creation instruction in the system. Compared to Lee et al.’s model, **K-LLVM** has a much simpler data layout and a concrete abstract machine to support different semantic behaviors including corner cases and side effects caused by the interaction of different instructions. Memarian et al. [36] provided two pointer provenance models for C/C++ languages and reconciled the ISO C standard. Similar to Lee et al.’s work, Memarian et al. focused on creating better pointer provenance models for C instead of investigating different C instruction behaviors through a concrete abstract machine. Without great effort, it is unclear how to build an abstract machine to support all LLVM IR instructions based on their model.

Other Large Language Specifications. **K-LLVM** is a formal and executable specification, of which many have been defined recently. Standard ML by Milner, Tofte, Harper, and MacQueen [37] is one of the most prominent and mathematical programming language specifications, whose formal and executable specifications were added to by Lee, Crary, and Harper [22], VanInwegen and Gunter [15], and Maharaaj and Gunter [31]. Blazy and Leroy [3] verified an optimizing compiler based on CLight in CompCert. Large language specifications have been defined in \mathbb{K} , including C [11], PHP [13], JavaScript [38], and Java [5]. A lot of work has been done on formalized specifications in Java and C#: Eisenbach’s formal Java semantics [9] and Syme’s HOL semantics [43] for Drossopoulou, the C# standard by Börger et al. [6], which is formally executable and uses abstract state machines [14], and the executable Java specification by Farzan et al. [12]. We cannot list all of the interesting examples of formalized language specifications in this paper for space reasons.

Our mechanized specification of **K-LLVM** shares many of the difficult challenges faced by the works described above and involves many new ones, due to the complex and dynamic nature of **K-LLVM**. They are detailed in later sections.

3 Background and Challenges

Below we discuss the major challenges that needed to be faced when developing **K-LLVM**. Additionally, we introduce briefly LLVM IR programs, **K-LLVM** and \mathbb{K} .

3.1 A Taste of LLVM IR Programs and Assumptions on LLVM IR

The LLVM language (LLVM IR) is a statically and strongly typed, assembly-like, Static Single Assignment (SSA) based language. It has undefined behaviors but the undefinedness is well documented. The LLVM language itself does not have operations or libraries to support multi-threaded behaviors, but LLVM IR's structure is highly related to the C/C++ library. LLVM IR basically assumes a runtime environment of C++. LLVM IR also contains a set of functions comprising an intrinsic library, in which part of the standard C library is included. It also relies on other functions in the `stdlib.h` header. For example, it needs dynamic memory management functions such as `malloc`, `realloc` and `free` to provide heap memory access, as well as functions dealing with the environment such as `abort`, `exit` and `system`. Furthermore, it needs functions listed in the `stdio.h` header to provide I/O support, as well as library functions from the Pthread and Pthread-mutex libraries to provide threading and mutual exclusion behaviors. These functions are not strictly part of the LLVM IR listed in the documentation but we define them anyway.

The current LLVM IR can be viewed as “C-”. Except function bodies, most features in C can be found in LLVM IR, such as global variables, `struct` datatypes, function headers and different flags for global variables or functions, etc. The main difference between LLVM IR programs and C programs are the function bodies, a.k.a. expressions. The LLVM IR expressions are register-based, SSA based and assembly-like. These features eliminate the undefinedness of the evaluation order in an LLVM IR program. We show some examples of LLVM IR expressions in Figure 1 to provide a taste of LLVM IR. These expressions are used throughout the whole paper. We believe that these expressions are enough to show the key features of LLVM IR and the construction of LLVM IR programs based on these expressions and other components (function headers, global variables and modules, etc) can be easily found in the LLVM documentation. This is also the reason we refer to these expressions as “programs” in the rest of the paper.

LLVM IR distinguishes local variables from global variables. Variables starting with the character `%` are local ones, while those starting with the character `@` are global. Global variables can only have a pointer type. Any number following the character `i` in LLVM IR, such as `i32` or `i1`, means an integer type declaration with the size of the bits. `i32*` refers to a 32-bit integer pointer type declaration. Instructions starting with the keyword `icmp` are the integer comparison operators. With the keyword `eq`, the instruction `%r8 = icmp eq i64 %r7, 47244640267` tests whether the value in the variable `%r7` and `47244640267` are the same and stores the result to the variable `%r8`. The “;” operation allows users to put comments after a line of code.

Program-A does several pointer arithmetic operations and memory operations. Several key observations about LLVM IR are made here. First, `getelementptr` is a memory address calculation operation and has an `inbounds` flag. No previous work has formally defined the behavior of flags of `getelementptr`. The definition of `inbounds` is hard because it not only

7:8 K-LLVM: A Relatively Complete Semantics of LLVM IR

```

Program-A :
1  %r1 = call i8* @malloc (i64 12)
2  %r2 = bitcast i8* %r1 to [3 x i32]*
3  store [3 x i32] [i32 0, i32 0, i32 0], [3 x i32]* %r2
4  %r3 = getelementptr inbounds [3 x i32], [3 x i32]* %r2, i64 0, i32 1
5  %u1 = getelementptr inbounds [3 x i32], [3 x i32]* %r2, i64 -1, i32 4 ;poison value.
6  %u2 = getelementptr inbounds i8, i8* %r1, i64 3
7  %u3 = load i8, i8* %u2
8  %u4 = ptrtoint i8* %u3 to i64
9  %u5 = add i64 %u4, 1
10 %u6 = inttoptr i64 %u5 to i8*
11 %u7 = load i8, i8* %u6
12 %r4 = bitcast i32* %r3 to [2 x i32]*
13 store [2 x i32] [i32 11, i32 11], [2 x i32]* %r4
14 %r5 = ptrtoint [2 x i32]* %r4 to i64
15 %r6 = inttoptr i64 %r5 to i64*
16 %r7 = load i64, i64* %r6 ;read back the two i32 array as an i64 value 47244640267.
17 %r8 = icmp eq i64 %r7, 47244640267
18 br i1 %r8, label %next, label %exit
19 next:
20 %r9 = inttoptr i64 100 to i32*
21 %r10 = getelementptr inbounds i32, i32* %r9, i64 0 ;poison value.
22 store i32 42, i32* %r9 ;unspecified behavior due to invalid pointer.
23 exit:
...

Program-B :
Thread-1 :
...
store atomic i32 42, i32* @x monotonic, align 1
%a = load atomic i32, i32* @y monotonic, align 1
...
Thread-2 :
...
store atomic i32 1, i32* @y monotonic, align 1
%b = load atomic i32, i32* @x monotonic, align 1
...

Program-C :
Thread-1 :
...
store i32 42, i32* @x
%a = load i32, i32* @y
...
Thread-2 :
...
store i32 1, i32* @y
%b = load i32, i32* @x
...

Program-D :
Thread-1 :
...
%a = load i32, i32* @x
%r = call i32 @pthread_create (i32 (*) @f, ...)
...
Thread-2 :
define i32 () @f {
store i32 1, i32* @x
return 0
}

Program-E :
%r1 = call i8* @malloc (i64 12)
%r2 = ptrtoint i8* %r1 to i32
%r3 = call i8* @printf (@x, i32 %r2)

```

■ **Figure 1** LLVM IR Example Programs.

affects the final result but also affects every intermediate result of computing the memory address. For example, in **Program-A**, `%u1` (line 5) is a poison value because we have `inbounds` in the `getelementptr`, and the second index is `i64 -1`, which makes the intermediate result out-of-bounds. Even though the final result is in bounds because we add back numbers, the `inbounds` still makes the final result a poison value. We talk about our definition of the `getelementptr` operation in Section 4.4. Second, as we mentioned in Section 2, LLVM IR views the main memory as having no type. We can store an array `[11, 11]` (line 13) and magically get back the `i64` value `47244640267` (line 16). This has effects on defining the **K-LLVM** type system, which will be explained in Section 4.1. Finally, executing **Program-A** in **K-LLVM** stops at the line 22. It is an unspecified behavior in LLVM IR to read data from a memory location pointed to by a pointer that was not properly created. This has not been properly defined by previous work, especially the definition of a memory operation combined with casting and pointer arithmetic operations. More details are in Section 4.4. **Program-B** and **Program-C** distinguish between a non-atomic and atomic memory operation. Thanks to our **K-LLVM** virtual machine definition, we are able to produce the race caused by two non-atomic operations in two different threads. Additional details are in Section 4. While maintaining sequential consistency, the execution of **Program-D** could result in a race on `@x` because of the special instruction execution order of LLVM, which the **K-LLVM** abstract machine models. More details are in Section 4.2. **Program-E** is an example for showing the usage of the \mathbb{K} symbolic execution engine in Section 5.

After reading the programs in Figure 1, questions about the memory locations and memory alignments may come to mind. Memory implementation is very complicated in real world programming languages. LLVM IR does not actually fix a special implementation of memory addresses. For simplicity, we assume in this paper that there is a one-to-one mapping from natural numbers to memory addresses, and a memory chunk is always in a range that can be defined between a left and a right integer bound. The memory addresses refer to conceptual memory byte data. Conceptual memory bytes are not actual byte data – details are in Section 4.3. LLVM IR also allows setting up alignments for different types, memory endianness and address space information by using `target datalayout`. Although we have implemented these features in **K-LLVM**, for simplicity, we assume in this paper that alignments, paddings for `structs` and address spaces never cause a problem in calculating memory addresses or type checking, and we assume little-endian byte-order. Finally, we assume that the heap size is infinite while the stack for each function is finite and has a maximum bound, and if a stack overflows in a thread, the whole system reaches an error state. We believe that assuming a max bound on the stack is an advantage of **K-LLVM** over previous formal semantics of LLVM IR. In the LLVM documentation, some stack intrinsic functions and function flags (`probe-stack/safestack`) indicates that function stacks has max bounds. The implementation of **K-LLVM** stacks is introduced in the description of the abstract machine (Sec. 4.2).

3.2 Challenges

Here we introduce some challenges that the development of **K-LLVM** faces.

Sheer Size of LLVM IR. The first challenge is the sheer size and precision of LLVM IR. With respect to instructions, LLVM IR has more than 60 operators and 100 intrinsic library functions. Some operators have complex rules or different requirements according to the input. For example, `store` operators can be either non-atomic or atomic, and atomic `store` operators have six different orderings. All of these require different semantic rules. The previous work only defined some of the operators, or some of their features. No previous work has defined the massive number of intrinsic library functions. **K-LLVM** defines all the LLVM operations and intrinsic functions. We handle this challenge through a special heavily testing strategy to define **K-LLVM** described in Sec. 5.

Subtlety of Well-formedness. In LLVM IR, the subtlety of various instructions and the well-formedness of instructions are often directly connected with the semantics of the instructions in a particular place in a given program. The syntactic nature of even a single instruction is determined by the semantic context. For example, the `getelementptr` operator allows indices to be integer local variables if the pointer input is an `array` pointer. However, if it is a `struct` pointer, LLVM IR requires the indices to be integer constants that can be statically reduced to integer values. These two types can be mixed together in a single usage of `getelementptr` in an LLVM IR program. Another example is that the input containing a decimal representation of a floating-point constant needs to be exact. This means that the value `1.1` cannot be a valid constant for floating-point operators in LLVM IR because it cannot be precisely represented by a finite floating point number, and LLVM IR requires the compilers to LLVM IR to round the float to a hexadecimal format. This is an error in Clang (the LLVM compiler).

Detailed Low-Level Features. As we mentioned in Section 1, it is not feasible to gloss over the details of LLVM IR’s low-level features, such as how to represent integers, floats and pointers. The effects are easily felt when we combine casting operations with memory operations. It is a common source of confusion among LLVM IR users, and thus, a common source of bugs. We also need to admit the fact that memory locations are highly related to integer behaviors; so converting pointers to integers, doing certain arithmetic on them, and converting them back to pointers are valid program exercises within a memory chunk created by a `malloc` operation. This brings us a big challenge. For example, in `Program-A` (Fig. 1), we cannot use pointer `%r9` to store data to the main memory (line 22), even if it is accidentally at the right range of a memory chunk, because `%r9` is not a valid pointer according to the LLVM IR pointer-aliasing rule. Defining a data structure to capture the behaviors covering all corner cases is one of the key contributions of **K-LLVM**. In addition, it is important to admit that the low-level structure of LLVM IR is based on bits and bytes; as well as the integer, float and pointer calculations are based on two’s compliments, IEEE 754, and integer pointer calculations.

Instructions Having Side-effects on Subsequent Instructions. Some instructions may cause side-effects on subsequent instructions depending on their behaviors. For example, in `Program-A` (Fig. 1), one can use the pointer `%r4` to access memory because it was a subsequent computation result of the pointer `%r1` from a `malloc` function, while `%r9` cannot be used to access memory data because it is from an integer constant. Defining these complicated side-effects requires new ideas. In addition, LLVM IR instructions can have very different requirements for different computer components. This complicates the design of different components of the **K-LLVM** abstract machine.

As we solved these challenges, we tried our best to define all language features in **K-LLVM**.

3.3 The \mathbb{K} Framework

\mathbb{K} [42] is a rewrite-based, executable semantic framework in which programming languages, type systems and formal analysis tools can be defined. **K-LLVM** is independent of \mathbb{K} . However, the implementation of **K-LLVM** in \mathbb{K} follows the mathematical definition closely, and some \mathbb{K} tools are useful for supporting the usage of **K-LLVM**. Once a language semantics is built, one can use *kompile* to see if it has static type problems and to generate an interpreter, so that users can use *krun* with the interpreter to test their semantics by actual concrete programs. *ksearch* allows users to search traces of multi-threaded programs based on the interpreter. The symbolic engine of *ksearch* and the program equivalence checker in \mathbb{K} allow users to compare two sets of traces from two different symbolically executed multi-threaded programs to see if their outputs are the same. Additional discussion is presented in Section 5.

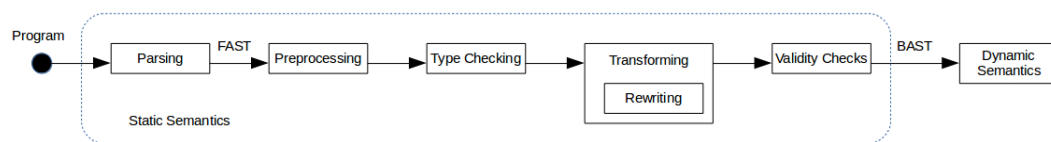
4 K-LLVM Semantics

In this section, we define the semantics of **K-LLVM**. It is divided into two parts: the **K-LLVM** static semantics (Sec. 4.1) and the **K-LLVM** dynamic semantics (Sec. 4.2 to Sec. 4.4). In this paper, we focus on parts of the descriptions of the static and dynamic semantics. We mainly discuss the general process and the type checking stage of the static semantics; as well as the general (sequentially consistent) abstract machine structure and memory operation specifications of the dynamic semantics. Other interesting details are in

our technical report and \mathbb{K} formalization [26]. Some important features of **K-LLVM** are based on VeLLVM [45]. For example, the **K-LLVM** formalization of SSA and Phi functions is very similar to the one in VeLLVM. The comparison of the work and **K-LLVM** is in Sec. 2.

4.1 K-LLVM Static Semantics

When giving the semantics of LLVM IR, **K-LLVM** uses two different ASTs, a front-end AST (FAST) and a back-end AST (BAST). The syntax of LLVM IR 6.0.0, which is documented in the website <http://releases.llvm.org/6.0.0/docs/LangRef.html>, is directly parsed into the FAST. We have formally defined the LLVM IR 6.0 syntax in \mathbb{K} , and it parses any LLVM IR program into the FAST format. **K-LLVM** static semantics refers to the LLVM IR behaviors that happen at compilation time. For an LLVM IR program, parsing is not enough to rule out unqualified programs. After parsing, a series of checks need to be performed on an LLVM IR program, including well-typedness, static single assignment, and well-formedness. The **K-LLVM** static semantics implementation applies these checks and rule out unqualified programs. It also translates a FAST program into a representation in the BAST format as first defined in [10], which is passed to the dynamic semantics for execution. Figure 2 depicts the phases in the **K-LLVM** static semantics.



■ **Figure 2** Static Semantics of **K-LLVM**.

Here we first sketch the functionality of each phase, and then focus on the type checking phase. More information can be found in the technical report [26]. The purpose of the preprocessing phase is to simulate the LLVM compilation steps that happen in the linkage time, including joining all modules from different files and dealing with global variables. The constant expression rewriting phase reduces LLVM IR constant expressions to values. After type checking, the transformation phase translates a program in FAST to a form in BAST. The validity checks phase applies well-formedness checks to the BAST program code, such as ensuring the code is in Static Single Assignment (SSA) form. We have proved the following theorem about the **K-LLVM** type system.

Type Checking. This step emulates the behaviors of LLVM IR type checking for the functions in LLVM IR modules. LLVM IR is a relatively strongly-typed language, and its type system is very straightforward. The **K-LLVM** type checking process is a complete implementation of the LLVM IR type system listed in its documentation. The input for the **K-LLVM** type checking function is a term and its type; the function outputs `true` if the term has been type checked and has the input type, and `false` otherwise. "Relatively strongly-typed" here means that the type system of LLVM IR guarantees the type preservation property, i.e., a typed value produced from a typed LLVM IR expression is compatible with the size of the value in runtime, and any later usage of the value will not result in a type error or size error if there is a move (usage). However, the program still has the chance of going wrong in the case of other problems, such as division by zero. In **Program-A** in Figure 1, every line of code except `store` and `br` instructions assigns a value to a variable. After type checking, each variable has a type. `%r1` has type `i8*` (line 1) and `%r2` has type

7:12 K-LLVM: A Relatively Complete Semantics of LLVM IR

`[3 x i32]*` (line 2). If we eliminate line 2 and replace the variable `%r2` in line 3 with `%r1`, the line results in a type error. In addition, there is also a chance that a correctly typed LLVM IR program is never executed since the execution of an LLVM IR program depends on the runtime environment setting. For example, The abstract machine in **K-LLVM** (Sec. 4.2) is parameterized by the function stack size. Users are free to set the size to 0, in which no program can be executed in any step.

```
%struct.RT = type {i8, [10 x [20 x i32]], i8}  
getelementptr inbounds %struct.RT, %struct.RT * %u, i64 0, i32 add (i32 1, i32 0), i32 %x
```

■ **Figure 3** A Type Example.

There are some tricky cases of the type system. In Figure 3, we show a `getelementptr` instruction on a `struct` type. For a `struct`, the value of the index for the `getelementptr` affects the type result of the final value of an instruction, because every position in a `struct` can have different type. Type checking a `getelementptr` relies on executing part of the semantics of the `getelementptr` arguments. That is why some index values of `getelementptr` that are associated to `struct` type positions are required to be inferred statically. This means that such positions can contain neither local nor global variables, even if a constant expression (no variables inside) is allowed. For other non-`struct` index positions, variables are allowed, such as the `x` `getelementptr` in Figure 3.

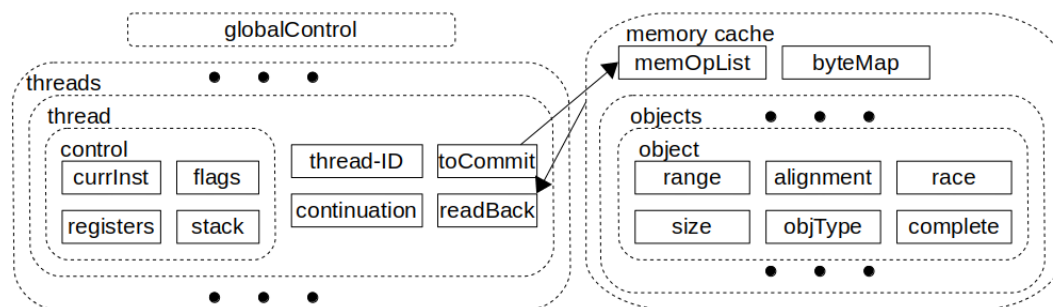
LLVM IR takes the view that values stored in the memory have no types, and that memory instructions will always produce values of the prescribed types. In fact, LLVM IR does not have a clear idea of main memory. It does not even have a built-in memory allocation instruction, instead, it relies on Standard-C library to provide such instructions. It basically assumes that the memory machine as a black box, and every memory request is **valid** as long as the size of the requested data matches the size of its type, the memory pointer is not out-of-range, and there is no race. In addition, one can have a correctly typed program where the result value produced by the program does not make sense. For example, loading an `i31` value from a heap field that is previously stored as an `i30` value is unspecified. To support the type system, **K-LLVM** assumes that each of the poison value and undefined value is implemented as a family of constructs, one for each type (ASTs as `undef (Type)` and `poisonValue (Type)`). Combining all these features of LLVM IR type system, we have shown the following type preservation theorem (the proof sketch is in the technical report [26]):

► **Theorem 1.** Assuming every load returning a value in a type prescribed in the load instruction, the program is well-typed by the **K-LLVM** type system, and the program executes at least one step, then every register and every return value of the program will be of the type assigned during the type checking.

The statement about the loading value in the theorem refers to that every load instruction reads a value that is previously stored with a proper type matching the load instruction, i.e., no having the case such as loading an `i31` value from a heap field that is previously stored as an `i30` value. The theorem assumes that every LLVM IR program can make a move, and it does not guarantee that the execution of a type-correct program has at least one move. After a program has been checked and transformed through the **K-LLVM** static semantics, the transformed BAST program is ready for execution by the **K-LLVM** dynamic semantics.

4.2 The K-LLVM Abstract Machine

As we mentioned in Section 1, the semantics of the execution of the LLVM IR programs in **K-LLVM** described is via an abstract machine. There are three reasons for this. First, it is a concise way to define all features and aspects of the LLVM IR semantics. LLVM IR is a programming language that connects different computer resources through many different instructions. The best way to model these different features is to design a computer-like mathematical entity which simulates them. Second, the abstract machine is designed to emulate real world computer components. Often, mathematical abstract machines are complicated and confusing. The **K-LLVM** abstract machine execution is easy for users to follow since they can relate it to real world computer components. Third, our abstract machine is modular; as a consequence, it is also extendable. In previous language semantics, designers either only define straight-line single-threaded instruction behaviors or only define a subset of all instructions with complete concurrent behaviors. Once concurrent behaviors are introduced, a single instruction's semantics can affect the whole semantic universe forcing designers to update all existing instruction semantics to handle any side-effects. The design of the **K-LLVM** abstract machine allows us to focus on designing one feature at a time in isolation. Additionally, because of the modular design, the abstract machine can be easily updated to support progressive concurrent features. For example, we update the byte-wise sequential consistency model in this paper to a model containing the full LLVM IR concurrency features in our corresponding technical report [26], without modifying a *single* instruction semantic rule, and only changing transition rules for describing how to maintain the execution order in the `continuation` and `toCommit` component of each thread.



■ **Figure 4** Component Relations in the **K-LLVM** Memory Layout Structure.

Figure 4 describes the overall structure of the **K-LLVM** abstract machine and the interactions of different components. The arrows show the direction of messages passing between the main components. A rounded dashed component means a program state entity that might contain other component structures, while a square component means a program state entity whose content is an integer, list, set, map, etc. The **K-LLVM** abstract machine is independent of the platform in which we implement the machine. At the top level, the abstract machine can be thought of as a set of threads communicating with a set of memory caches, and a global control unit provides global information for threads. As a simplifying assumption to achieve byte-wise sequential consistency, we assume the memory cache set is a singleton set, so we will refer to this cache as the memory cache in the rest of the paper. The `globalControl` component represents the global control unit containing several sub-components storing information about threads, such as thread identifier calculation, thread final states and mutex lock information. We will see an example of using this information in Section 5. There are several components in each thread as shown in the left side of Figure 4.

In Section 4.1, we said that a LLVM IR program is compiled to a list of BAST control flow graphs (CFGs) for execution. The `continuation` component represents a dynamically executing CFG; it contains a sequence of dynamic basic blocks of instructions to be executed. A thread executes one instruction at a time, i.e., the first instruction in the first block. Thread execution is modeled by consuming instructions as they are executed and possibly inserting a new basic block after the current block during loop execution.

For each thread, the `control` component includes `registers`, a `stack`, `flags`, and `currInst` components. The `registers` component is a map from local variables to values. We introduce how we represent values in the next section. The `stack` component records function call stack frames for context switching in LLVM IR based on `call` and `return` instructions. Each stack also contains fields for local memory allocations in a function directed by the `alloca` instruction. The **K-LLVM** stack implementation is not a simple mapping. Each **K-LLVM** function stack has a maximum allowed allocation space, and stack overflow leads to an error state. Every time when a function is called, a memory range is actually created in the heap for storing the function stack information. This implementation allows us to implement some LLVM IR flags such as “`inalloca`”, and also some extra tools built on top of **K-LLVM** (as a future work) to track stack buffer overflows such as `AddressSanitizer` and `SAFECODE`. The `flags` component contains the set of function header flags describing the function that is currently executing. For example, the `readonly` flag tells the LLVM compiler that the function will never produce memory write operations, and this need to be reflected in the execution semantics; see Section 4.4 for a complete semantics. The `currInst` component contains a dynamic block number and instruction number pair representing the unique identifier for the currently executing instruction. Dynamic block numbers are basically timestamps and uniquely identify each execution of a basic block; when a new basic block of instructions is put into the `continuation` component, a new such number is associated with the block. Instruction numbers can be assigned statically, e.g., using textual order in the LLVM IR file. For example, the numbers on the left side of `Program-A` (Fig. 1) are a possible instruction numbering.

The `currInst` pair allows us to modularly add new concurrent behaviors to the **K-LLVM** abstract machine. Even though our model assumes byte-wise sequential consistency in this paper, the machine has potential for additional concurrent behaviors. When dispatching a memory instruction, a thread need not wait for the instruction commit before proceeding. For example, a thread will not wait for a `load` instruction to write values to `registers`. Instead, it moves on and marks the specific register as unavailable. If the next instruction needs the register value, the thread component blocks. Otherwise, the thread continues to execute instructions. The `currInst` pair identifies a specific instruction and corresponding register during write back. The example `Program-D` (Fig. 1) shows how this feature can affect program execution in practice. Without this mechanism, the `load` instruction in `Thread-1` always happens before the `store` in `Thread-2`. With this mechanism, an observer can observe the value 1 or even a race on `@x`. This example is the motivation of having the abstract machine in **K-LLVM** even though its memory model assumes byte-wise sequential consistency in this paper. **K-LLVM** is mainly used to verify LLVM compiler steps, and verifying programs containing library functions is a key verification component. The `pthread_create` function in `Program-D` is a library function and its functionality should contain fences to prevent the behavior of executing `Program-D` described above. The abstract machine mechanism in **K-LLVM** allows to prove that a particular implementation of `pthread_create` does not have harmful behaviors like the one above; whereas otherwise we do not have a mechanism to verify such library function usage in a program.

The `toCommit` and `readBack` components in a thread are to deal with memory instructions, and they also act as interface communicating with the memory cache. From the memory point of view, all it knows about memory requests from each thread are from the two components. In this sense, they belong to the memory model of the **K-LLVM** abstract machine, even though they are located in each thread. The `toCommit` component is implemented as a queue that receives memory operations from `continuation` and then sends them to the memory cache in order. The `readback` component is implemented as a map and represents the intermediate step of getting back a value from a memory-read from the memory cache and assigning the value to registers. These components are needed to distinguish between memory instructions and their corresponding execution. Another reason is the need to simulate the difference between the non-atomic and atomic memory operations in LLVM IR. LLVM IR assumes that each non-atomic memory write or read operation accesses a single byte of data in the memory cache at a time, while an atomic operation accesses several bytes at once. By breaking down the execution of non-atomic `store` and `load` instructions into possibly several memory operations, we are able to capture potential races in a multi-threaded program.

The memory cache has a fixed structure in **K-LLVM**, which is listed on the right side of Figure 4. The `memOpList` component stores the memory operations from different threads, in order to allow the interleaving of memory operations from different threads. The `byteMap` component is a function that maps a memory location to a byte of data. A memory write operation in **K-LLVM** stores an array of bytes in the `byteMap` component. While `byteMap` represents the entire memory cache, a memory chunk refers to a continuous memory region in `byteMap` and is allocated by a global memory initialization or local memory allocation. An `object` component stores metadata for a specific memory chunk. Each `object` contains a `range` component indicating the range of the chunk in the whole memory domain (as keys of `byteMap`), an `alignment` component with alignment information, a `size` component with the size of the chunk in bytes, and an `objType` component indicating if the memory chunk is static or not. The `complete` and `race` components are used to record the status of the operations accessing the memory chunk. According to the LLVM IR documentation, non-atomic memory operations should access a memory range one byte at a time. When a non-atomic memory operation is accessing a memory chunk at the same time as another memory write operation, a race occurs, and the result is `undef`. The `complete` and `race` components are used to record this status and give the result. The implementations of the `byteMap` and `object` components are used to represent the low-level memory layout structures in LLVM IR. The reasons to have these components are indicated in Sec. 2. We summarize them here. The key requirement of having these components in **K-LLVM** is to enable a "heavy-weight" pointer provenance model that can carry provenance information for pointers in every place under the multi-threaded environment, while keeping components independent and only communicating through observable "official" channels. We believe that storing metadata on a per-chunk basis is the best way to implement the LLVM IR memory layout model to maximize the concurrent memory access behaviors allowed by LLVM IR.

We have briefly described the different components of the **K-LLVM** abstract machine above. The details of the implementations of each component can be found in our implementation [26]. In the following sections, we will introduce some detailed implementation aspects related to memory accesses. The full LLVM IR concurrency model can also be found in the technical report [26].

4.3 K-LLVM Data Layout

In this and the next sections, we introduce a portion of the **K-LLVM** abstract machine in depth, especially, the components and rules related to executing memory related instructions. The manner in which data layout and memory layout are implemented in **K-LLVM** facilitates

the precise semantics of many language features of LLVM IR while maintaining a concise abstract machine for the execution semantics. In this section, we introduce the implementation of register and memory location values in **K-LLVM** and example rules using these values. The need for two different kinds of values arises as from the fact that memory only sees values as a sequence of bytes, while instructions see registers as holding compound data. We describe these two kinds in Figure 5, and we also show some example rules using these data. In Figure 5, rules connected by a \Rightarrow operator mean that the transition from the left hand side to the right hand side happens in the beginning of a `continuation` component. There is an implicit rule saying that every transition happening in the beginning of a `continuation` also happens globally. More complex transition rules are introduced in Fig. 7. The `add` and `icmp eq` are instructions in LLVM IR appearing here in the concrete syntax.

The `undef` value for a *Bit* datum exists due to undefinedness of LLVM IR. In LLVM IR, if an integer that is not a multiple of the length of a byte (like a 23-bit integer), and is stored to the memory, then the values for the extra bits generated during the process are undefined (`undef`). A memory location value is implemented by the *Byte* type. In addition to having eight *Bit* data, each *Byte* datum contains a range attribute (*Range Option*) and a flag attribute (*Range State*). If a *Byte* datum represents a part of a pointer, the range attribute is the left and right edges of the memory range to which the pointer points, and if not then `none`. If a *Byte* datum represents a part of a pointer, and the pointer is the result from a `getelementptr` instruction with a `inrange` flag, the flag attribute is the left and right edges of the memory range that the `inrange` flag defines. If the pointer does not come from a `getelementptr` instruction, the flag attribute is `none`. If a `getelementptr` generates an error due to mixing of `inrange` flags, the flag attribute records the error. We will see more about the `inrange` flag of a `getelementptr` in the paragraph describing `store` instruction semantics below. We want to have these two attributes associated with a *Byte* datum because we want to provide pointer provenance, so that when a pointer is cast to an integer or stored to the memory cache, it does not lose side-effect information, such as what is the memory field the pointer points to. The real data structure of *Byte* data in **K-LLVM** has more fields including information about block address information, endianness, and if a pointer datum is pointing to a heap, stack, or static constant memory chunk. For simplicity, we do not include them here, and assume the bytes are in little-endian format. We also assume no distinction between heap and stack pointers here, even though we have distinct implementations for each in **K-LLVM**.

```

Bit ::= 1 | 0 | undef   Range ::= range(Nat , Nat)   'a State ::= Error | 'a Option
Byte ::= byte(Bit List , Range Option , Range State)
Loc ::= loc(Bit List , Type , Range Option , Range State)
Int ::= intLoc(Bit List , Type , Range Option , Range State)
Float ::= floatLoc(Bit List , Type , Range Option , Range State)
(a)  add T intLoc(X, A1, B1, C1), intLoc(Y, A1, B2, C2)
      ⇒ intLoc(bitAdd(T, X, Y), A1, judge(B1, B2), judge(C1, C2))
(b)  icmp eq T loc(X, A1, B1, C1), loc(Y, A1, B2, C2) ⇒ intLoc([X = Y], i1, none, none)

```

■ **Figure 5** Memory Data Structure.

For register values, we only introduce integer, float and pointer values here. The description of other register values can be found in the **K-LLVM** semantics implementation [26]. Any of the integer (*Int*), float (*Float*) or pointer (*Loc*) data contains a *Bit* list, a *Type* field representing the type of the datum, a range attribute and a flag attribute. The *Bit* list represents the binary format of the value for the datum being either an integer, float (in

the IEEE 754 format) or memory address. The size of the list is equal to the size of the integer/float/pointer type defined for the data (the pointer size is parameterized in **K-LLVM**). We assume that all integer, float and pointer arithmetic is based on the computation of binary representations, even though we might show decimal representations in some examples in this paper for presentation purpose. The range and flag attributes have meaning that is closely related to the ones in a *Byte* datum, as we will explain below.

The reason for making the register and memory data structure so complicated is that **K-LLVM** covers the relatively complete semantics of LLVM IR including corner cases of not only the individual instruction semantics but also the interactions between casting, arithmetic and memory related instructions in LLVM IR. Hence, the pointer provenance information needs to be available both in the threads and the memory cache. In **K-LLVM**, the provenance information is stored in the value representation to enable three features of LLVM IR that require execution decisions based on the past history of the value. First, there are flags (*inrange*), which require the possibility of turning the transition state to an error state in executing a memory instruction long after the computation of a `getelementptr` with the flags. Second, a pointer is valid for accessing a memory datum if and only if it is created from a non-free memory allocation, or it is the result of a finite number of memory computations based on a non-free memory allocation pointer, and its pointing memory field is within the memory range of the allocated chunk. Third, an error should be detected when an execution is accessing memory data by a pointer cast from an integer value whose calculation never involves values cast from pointers, even if the integer has the same value as the memory address of a valid pointer.

The two rules (a) and (b) in Fig. 5 give an example describing how an arithmetic instruction is executed in **K-LLVM** based on the data structure described above. In evaluating an LLVM IR `add` instruction (rule (a)), the value computation happens between the *Bit* lists of two data (`bitAdd` adding two binary numbers together). The function `judge` merges two range or flag attributes from possibly two different data that possibly come from two pointer sources. The `judge` details are in the actual **K-LLVM** semantics implementation [26]. Here, we give some interesting examples. If a pointer is cast to a integer constant (with the range attribute $[L, R]$) and added to another integer constant (with the range attribute `none`), the `judge` produces a memory range from the pointer in the range attribute of the result datum. If the two range attributes of two `intLocs` have two different memory ranges (like $[L, R] \neq [S, T]$), the judged result is `none`. If two flag attributes of two data have two different memory ranges, in this case, `judge` produces an error state in the flag attribute of the result datum; and if the result datum is further turned into a pointer, and is used to read memory data, the program results in unspecified behavior. Rule (b) gives an example of a comparison instruction that discards the pointer information and produces a pure 1-bit integer constant. Depending on the instruction, including the nature of its arguments, pointer information might or might not be transmitted along with the result of the calculation.

4.4 Sample Instruction Semantics

In this section, we introduce semantic rules supporting memory related instructions in **K-LLVM**. The set of memory related instructions we select to describe here contains LLVM IR casting, address calculation (`getelementptr`) and memory instructions, as well as memory related flags on the function headers. **K-LLVM** is the first formal semantics to cover all behavioral aspects (under byte-wise sequential consistency) of this set, including the side-effects due to interactions between different instructions inside or outside of the set. Under the byte-wise sequential consistency assumption, the behaviors of different orderings in an atomic memory operation collapses to the behavior of the sequentially consistent (`seqcst`)

ordering. It is worth noting that there are cases when an instruction can go to an unspecified behavior or other error states. We will not list all of those rules here, although we have defined them in **K-LLVM**. Interested readers may get more details from the **K-LLVM** semantics [26].

Casting Instructions. Here we describe the semantics of `inttoptr` and `bitcast` as the highlights of the **K-LLVM** semantics of casting instructions in Figure 6. The other casting instructions are implemented in a similar manner. Before **K-LLVM**, no complete interpretation for the LLVM IR casting operations existed, especially one supporting casting between integers or floats and pointers. These casting instructions are hard to define because the resulting values can vary depending on the program context for the values of the instructions.

- (a) $\text{inttoptr}(\text{intLoc}(X, T1, B, C), T2) \Rightarrow \text{loc}(\text{trunc}(X, \text{sizeof}(T1) - \text{sizeof}(T2)), T2, B, C)$
 $\text{if } \text{sizeof}(T1) \geq \text{sizeof}(T2)$
- (b) $\text{inttoptr}(\text{intLoc}(X, T1, B, C), T2) \Rightarrow \text{loc}(\text{addZero}(\text{sizeof}(T2) - \text{sizeof}(T1))@X, T2, B, C)$
 $\text{if } \text{sizeof}(T1) < \text{sizeof}(T2)$
- (c) $\text{bitcast}(\text{Label}(X, T1, B, C), T2) \Rightarrow \text{rebuild}(X, T2, B, C)$
 $\text{if } \neg \text{isPointerType}(T1) \wedge \text{Label} \in \{\text{intLoc}, \text{floatLoc}\}$
- (d) $\text{bitcast}(\text{loc}(X, T1, B, C), T2) \Rightarrow \text{loc}(X, T2, B, C)$

■ **Figure 6** Casting Rules.

In Figure 6, rules (a) and (b) describe the semantics of `inttoptr`. The main idea is to replace the type attribute of the source `intLoc` with the target type. If the target type size is smaller than (or equal to) the source one, the semantics truncates (using the `trunc` function) the bits (represented by X as a list) by the difference of the sizes of the two types starting from the most significant bit. Otherwise, we create a list of 0 bits, whose size is the difference between the two type sizes, by using the `addZero` function. We place the bits in front of the source bit list (variable X). For example, in **Program-A** (Fig. 1), we assume that the code is running in a 32-bit machine and variable `%r5` has the value represented by `intLoc(X , i64, B , C)` in line 15. The code tries to convert the `%r5` value to a pointer. The final result pointer can be represented by `loc(X' , i64*, B , C)` by taking the right-most 32-bits from X and changing the constructor from `intLoc` to `loc`.

Rules (c) and (d) describe the much simpler dynamic semantics of `bitcast` instructions. Besides the memory data layout, the **K-LLVM** type system also contributes to the simplicity. Once we find out that $T1$ is not a pointer, we can immediately infer that $T2$ is also not a pointer because LLVM IR only allows pointer to pointer or non-pointer to non-pointer `bitcast`. Thus, the rule (c) should take the bits (variable X) with additional attribute information and distribute them to form a corresponding value with respect to the type $T2$, which is what the function `rebuild` does. For example, if we `bitcast` an `i24` integer (as `intLoc(X , i24, B , C)`) to a three `i8` integer array [`3 x i8`], the 24-bit list X is cut into three equal parts ($X1$, $X2$ and $X3$), so we have an array with three elements of the format `intLoc(Y , i8, B , C)` where Y can be either $X1$, $X2$ or $X3$. Alternatively, if a `bitcast` sees a `Loc` datum, it is immediately inferred that the casting is between two pointers, and the only effect is the updating of the source type $T1$ with the target type $T2$.

The Semantics of `getelementptr`. A `getelementptr` instruction is a memory address calculation whose main idea is to calculate a memory address value based on a sequence of indices. Section 3.1 touches on one of the special cases of `getelementptr` semantics. The main idea of `getelementptr` is similar to the one in Zhao's work [45]. It uses a sequence of

indices of different types to walk incrementally into a data structure layout to calculate a pointer to the sub-component found at the end of the path the indices describe. Here, we focus on one particularly important feature of the instruction, the keyword `inbounds`, which is a flag applied on the computation results of a `getelementptr` instruction. For this flag, LLVM IR requires all the intermediate and final computation results on the address of the input pointer are within a valid range of the allocated object pointed to by the address. In **K-LLVM**, we implement this with the address computation function `calGEP`. The function calculates a new address value by adding multiplication results of the index and type size to the input address, one adding at a time. In each step, before the calculation, the function first checks if the input address is within the range indicated by the range attribute of the input pointer. After we compute the final address result, we also check if the memory chunk pointed to by the input pointer still exists. For example, line 4 of **Program-A** (Fig. 1) is a `getelementptr` instruction, and it is executed successfully in **K-LLVM**. However, if a memory-free for the input pointer `%r2` is added before the `getelementptr`, the `inbounds` flag makes the instruction result in a poison value, because the memory chunk pointed to by `%r2` does not exist anymore. As another example of an `inbounds` flag, executing line 5 of **Program-A** highlights how a poison value can be produced from a `getelementptr`. The index `i64 -1` makes an intermediate computation result out-of-bound, so variable `%u1` gets a poison value. Another example is to execute line 21 of **Program-A**. The execution of this `getelementptr` fails the `inbounds` check because its input pointer has range attribute `none`, so variable `%r10` results in a poison value. There is also an `inrange` flag in a `getelementptr` instruction. This flag has subsequent effects on memory instructions after the `getelementptr`. The flag information is carried as the flag attribute in the pointer derived from the `getelementptr` so that the succeeding memory instructions can use it. We will introduce its semantics in the next section.

The store Semantics. We only introduce the **K-LLVM** `store` memory instructions here; the other memory instructions are implemented in a similar manner. **K-LLVM** fully implements the semantics of `stores` under the byte-wise sequential consistency assumption. Specifically, **K-LLVM** distinguishes the non-atomic and atomic `store` instructions by breaking the execution of an memory instruction into three different stages, as shown in Figure 7. As we mentioned, we do not list negative rules, such as configurations going to an error state when a `store` is performing a `write` operation in the memory cache, when the memory chunk has already been freed by another thread. The rules in Figure 7 are simplified versions of the actual **K-LLVM** rules. The information and handling about address spaces and memory alignments is not mentioned here. In fact, the construct `write` has several fields than one shown in the figure. On the other hand, these rules are non-trivial, and they have enough functionality to show manner in which the **K-LLVM** abstract machine distinguishes between the behaviors of atomic and non-atomic `store` instructions.

In Figure 7, the *Exp* type represents an instruction that involves in the computation in a continuation component (Ψ in Fig. 7). We uses `store` and `atomicStore` constructs in Figure 7 that are different from the LLVM IR concrete syntax. They are BAST format transformed from an LLVM IR `stores` instruction in their simplified form here. Each of them has three fields. The first represents the type of the value; the second is the value to store in the memory cache and the third is the memory pointer. The `write` construct represents the memory operation that a thread uses to communicate with the memory cache and the memory cache uses to perform memory events. When a `store` is executed in the continuation component (Ψ), a list of `writes` are generated in the `toCommit` component

$$\begin{array}{l}
\text{Key} ::= (\text{Nat}, \text{Nat}, \text{Nat}) \quad \text{Byte List} ::= \text{toBytes}(\text{Exp}, \text{Nat}) \\
\text{Exp} ::= \text{store}(\text{Type}, \text{Exp}, \text{Loc}) \mid \text{atomicStore}(\text{Type}, \text{Exp}, \text{Loc}) \mid \text{write}(\text{Key}, \text{Nat}, \text{Nat}, \text{Byte List})
\end{array}$$

(a)	$\frac{[X, X + \text{sizeof}(T)] \subseteq [L, R] \wedge [X, X + \text{sizeof}(T)] \subseteq [L1, R1] \wedge \text{readonly} \notin \Theta}{\begin{array}{l} (TID, (BID, IID), (\text{store}(T, V, \text{loc}(X, B, \text{range}(L, R), \text{range}(L1, R1))) :: \Psi), \Delta, \Theta) \\ \Rightarrow (TID, (BID, IID), \Psi, \\ \Delta @ \text{genWrites}(\text{toBytes}(V, \text{sizeof}(T)), (TID, BID, IID), X, \text{sizeof}(T)), \Theta) \end{array}}$
(b)	$\frac{[X, X + \text{sizeof}(T)] \subseteq [L, R] \wedge [X, X + \text{sizeof}(T)] \subseteq [L1, R1] \wedge \text{readonly} \notin \Theta}{\begin{array}{l} (TID, (BID, IID), (\text{atomicStore}(T, V, \text{loc}(X, B, \text{range}(L, R), \text{range}(L1, R1))) :: \Psi), \Delta, \Theta) \\ \Rightarrow (TID, (BID, IID), \Psi, \Delta @ [\text{write}((TID, BID, IID), X, 1, \text{toBytes}(V, \text{sizeof}(T)))]), \Theta) \end{array}}$
(c)	$\begin{array}{l} \left(\{ (TID, \text{CurrInst}, \Psi, E :: \Delta, \Theta) \cup \text{Threads} \}, (\kappa, \text{Rest}) \right) \\ \Rightarrow \left(\{ (TID, \text{CurrInst}, \Psi, \Delta, \Theta) \cup \text{Threads} \}, (\kappa @ [E], \text{Rest}) \right) \end{array}$
(d)	$\frac{\text{Addr} \in [L, R] \wedge \neg \text{isRace}(\text{Key}, \alpha)}{\begin{array}{l} (\text{write}(\text{Key}, \text{Addr}, 1, V) :: \kappa, \Gamma, \{ ([L, R], \alpha, \text{Rest}) \} \cup \Omega) \\ \Rightarrow (\kappa, \text{updateMap}(\Gamma, \text{Addr}, V), \{ ([L, R], \alpha, \text{Rest}) \} \cup \Omega) \end{array}}$
(e)	$\frac{\text{Size} > 1 \wedge \beta(\text{Key}) = \text{none} \wedge \text{Addr} \in [L, R] \wedge \neg \text{isRace}(\text{Key}, \alpha)}{\begin{array}{l} (\text{write}(\text{Key}, \text{Addr}, \text{Size}, V) :: \kappa, \Gamma, \{ ([L, R], \alpha, \beta, \text{Rest}) \} \cup \Omega) \\ \Rightarrow (\kappa, \text{updateMap}(\Gamma, \text{Addr}, V), \{ ([L, R], \alpha \cup \{\text{Key}\}, \beta[\text{Key} \mapsto 1], \text{Rest}) \} \cup \Omega) \end{array}}$
(f)	$\frac{\text{Size} > 1 \wedge \beta(\text{Key}) = \text{Size} - 1 \wedge \text{Addr} \in [L, R] \wedge \neg \text{isRace}(\text{Key}, \alpha)}{\begin{array}{l} (\text{write}(\text{Key}, \text{Addr}, \text{Size}, V) :: \kappa, \Gamma, \{ ([L, R], \alpha, \beta, \text{Rest}) \} \cup \Omega) \\ \Rightarrow (\kappa, \text{updateMap}(\Gamma, \text{Addr}, V), \{ ([L, R], \alpha \setminus \{\text{Key}\}, \beta[\text{Key} \mapsto \text{none}], \text{Rest}) \} \cup \Omega) \end{array}}$

■ **Figure 7** Memory Store Rules.

(Δ in Fig. 7). They have the same group ID represented as a *Key* type that is a triple of the thread ID, dynamic block number, and instruction number of the `store`. A `write` also has other fields: a natural number representing the memory address value, another natural number representing the total size of `writes` from the same *Key*, and a list of bytes to write to the memory. The total size is the same for different `write` operations with the same *Key*. It is both the size of the list of `writes` generated by a non-atomic `store` and the size of bytes of the value to write to the memory cache. An `atomicStore` generates a singleton `write`.

Before we describe the rules in Figure 7, some conventions are worth noting. Without special greek letter illustrations on different components, a name of a component with its first character capitalized is the variable representing the component in all rules (e.g. *CurrInst* for the `currInst` component, and *Threads* for the `threads` component). The variable *Rest* appearing in some rules in Figure 7 (and Fig. 8) represents the rest of components in a `thread` or `object` component, which do not involve in the computation of the rules. As we have said in Section 4.2, the **K-LLVM** abstract machine is for a set of threads communicating with a single memory cache. The `globalControl` component is omitted in the computation here, since we do not need it. Based on these assumptions, we define a transition state to be a pair of a set of threads and a memory cache: $(\text{Threads}, \text{Memory})$. A single thread contains five components related to memory instructions: `thread-ID`, `currInst`, `continuation` (Ψ), `toCommit` (Δ) and `flag`. For simplicity, we assume that a thread only contains these five components in this section; Also, we assume that the memory cache only contains three components: the `memOpList` (κ), `byteMap` (Γ) and `objects` (Ω) components. The `objects` component (Ω) contains a set of `object`. Only three sub-components (`range`, `race` (α) and `complete` (β)) in the `object` are related in defining the semantics of `store`. The math

inclusive range $[A, B]$ represents a set of natural number sequencing from the number A to the number B inclusively. Finally, there are implicit rules omitted in Figure 7, suggesting that transitions happening in a thread or the memory cache also happens globally.

Rules (a) and (b) in Figure 7 describe how an atomic `store` and non-atomic `store` generate a list of `write` operations that are pushed to the `toCommit` component (Δ) whose job is to convey memory operations to the memory cache. The basic idea is to create a list of `writes` at the end of `toCommit` (Δ) when we have a `store` in the head of `continuation` (Ψ). The two rules describe the cases when an `inrange` flag is present in the flag attribute of the input pointer. In such cases, to execute a `store` not only requires for the address value to be within the range indicated in the pointer but also for it to be in the range carried by the `inrange` flag; otherwise, the whole system results in an unspecified behavior state. In **K-LLVM**, there are rules similar to rules (a) and (b) dealing with pointers without `inrange` flags derived by removing the checks for the `inrange` edges from rules (a) and (b). Since we will use these rules in an example, we call them rules (ax) and (bx) to distinguish them from rules (a) and (b). The function `toBytes` splits a value into a list of bytes (AST in Fig. 7). The list size is defined by its natural number argument. Its functionality is similar to the `rebuild` function to turn a value into a list of elements. The only difference is that `toBytes` creates a list of bytes instead of values in the case of `rebuild`. The function `genWrites` takes a list of bytes, a `Key` datum, a memory address, and the size of the byte list, then generates a list of `writes` by distributing a byte at a time from the byte list, and associates each byte with a memory address and other attributes. The address value is selected in sequence from the address range between the address and the address plus the size. Rule (b) is for dealing with atomic `stores`. The key difference is that it only generates a singleton `write` containing the full value to be stored instead of a list. Rule (c) allows the head element in the `toCommit` component (Δ) of a thread to move to the tail position of the `memOpList` (κ) in the memory cache.

Rules (d), (e) and (f) deal with different situations of correctly committing a `write` to the `byteMap` (Γ). The `complete` component (β) in (e) and (f) is a map from a `Key` to a natural number indicating how many `writes` have been committed to `byteMap` (Γ) since the first `write` with the `Key`. The `Key` marks a single instruction and `complete` allows tracking the process of the writes entailed by the instruction. To detect races, the `race` component (α) contains `Keys` indicating every `Key` occupying the memory chunk (`object`) represented by the `range` of the `object` component. The variable `Size` represents the number of `writes` from the same `Key`, i.e. the same `store` instruction. All rules (d), (e) and (f) need to satisfy two side conditions. The first one is the condition $Addr \in [L, R]$ to locate a specific `object` in the `objects` component (Ω) by comparing `Addr` with the range of the `object` (L and R). In **K-LLVM**, an `object` is created by a memory allocation; thus, the ranges of `objects` (Ω) are always disjoint. Any address (e.g. `Addr`) within a range (e.g. $[L, R]$) can be a key to locate the range, which in turn locates an `object`. The second condition is to check if a `Key` is in race with other `Keys` in `race` (α) by the function `isRace`. The function `isRace` checks if the `race` component (α) for the `object` pointed to by the memory address value (`Addr`) has been occupied by another `Key`. If `Size` is 1 (rule (d)), the `write` represents an atomic memory `store`, and writes a list of bytes (V) to `byteMap` (Γ) using the function `updateMap`. The function updates a range of bytes to corresponding range of addresses in `byteMap` (Γ). Rule (e) is executed if two other conditions are satisfied: the `Size` is not 1 and no `write` for this `Key` has yet completed. In such case, rule (e) writes a list of bytes to `byteMap` (Γ) and updates the information in the `race` (α), and initializes the `Key` in the `complete` component (β). Rule (f) represents the finish of the execution of a

non-atomic `store` in the memory cache. In such cases, we remove the appearance of the entities represented by variable `Key` in the `race` (α) and `complete` (β) components. We also need to update `byteMap` (Γ) with the final `write` term. Besides rule (e) and (f), another rule not listed here deals with the case when $\beta(\text{Key})$ does exist and is less than `Size - 1`. In this rule, we continue to write a byte to the `byteMap` component (Γ) without touching the `race` component (α) and incrementing the `complete` component (β) for `Key`.

$$\begin{aligned}
& \text{(s)} \left(\left\{ \left(\varphi, (1,13), (\text{store}([2 \times \text{i32}], [11,11], \text{loc}(100, [2 \times \text{i32}]^*, \right. \right. \right. \\
& \quad \left. \left. \left. \text{range}(96, 108), \text{none}) \right) : \Psi, \square, \emptyset \right) \right\} \cup \Xi, \left(\square, \Sigma, \{ ([96,108], \emptyset, \emptyset, \Upsilon) \} \cup \Omega \right) \right) \\
& \Rightarrow \left(\left\{ \left(\varphi, (1,13), \Psi, (\text{write}((\varphi, 1,13), 100, 8, \text{byte}(B, \text{none}, \text{none})) : \Delta), \emptyset \right) \right\} \cup \Xi, \right. \\
& \quad \left. \left(\square, \Sigma, \{ ([96,108], \emptyset, \emptyset, \Upsilon) \} \cup \Omega \right) \right) \\
& \Rightarrow \left(\left\{ \left(\varphi, (1,13), \Psi, \Delta, \emptyset \right) \right\} \cup \Xi, \right. \\
& \quad \left. \left([\text{write}((\varphi, 1,13), 100, 8, \text{byte}(B, \text{none}, \text{none}))], \Sigma, \{ ([96,108], \emptyset, \emptyset, \Upsilon) \} \cup \Omega \right) \right) \\
& \Rightarrow \left(\left\{ \left(\varphi, (1,13), \Psi, \Delta, \emptyset \right) \right\} \cup \Xi, \right. \\
& \quad \left. \left(\square, \Sigma[100 \mapsto \text{byte}(B, \text{none}, \text{none})], \{ ([96,108], \{ (\varphi, 1,13) \}, \{ (\varphi, 1,13) \mapsto 1 \}, \Upsilon) \} \cup \Omega \right) \right) \Rightarrow \dots \\
& \text{(t)} \left(\left\{ \left(\varphi, (1,22), (\text{store}(\text{i32}, 42, \text{loc}(100, \text{i32}^*, \text{none}, \text{none})) : \Psi, \square, \emptyset) \right) \right\} \cup \Xi, \text{Rest} \right) \\
& \Rightarrow \left(\text{error unspecifiedBehavior} \right)
\end{aligned}$$

■ **Figure 8** Memory Store Example Configuration Transitions.

As an example of applying the `store` rules, we focus on the `store` instruction in line 13 of `Program-A` (Fig. 1). Group (s) in Figure 8 represents the computations for executing the first few steps of the `store` instruction. In these diagrams, we show the computations as transitions from one state to another. Each transition state is a tuple of a thread set and the memory cache. In threads, Ξ represents all threads that are not involved in the computation. The thread we care about has a thread ID φ . We assume that the (1,13) in the first state after the label (s) represents the `currInst` pair. In the `continuation` component, we have the `store` instruction of line 13 (Fig. 1) on the top of the computation, and Ψ represents the rest of the computations in `continuation`. For simplicity, we assume that the `toCommit` and `flags` components are empty for thread φ , so they have the values \square and \emptyset , respectively. In the memory cache, for simplicity, we assume that `memOpList` is empty, `byteMap` is represented by the variable Σ . The memory cache contains some objects. The Ω in Fig. 8 represents objects not related to this `store` computation, and there is an object with range value [96,108] that matters in this computation (Let’s assume that [96,108] is the memory range created previously). We also assume that the current `race` and `complete` components are both empty (an empty set and empty map). Υ represents the rest of the components in the object that is not involved in the computation. The to-store data for the `store` operation is an array of type `[2 x i32]` and value `[11,11]`. Here, we show these data in decimal formats. In the real **K-LLVM** abstract machine, they should be in the binary format. In this example, we assume that the memory pointer address is a natural number 100, and the range of the memory chunk pointed to by the pointer is in the range [96,108].

By applying rule (ax) above, we get a new transition state after the first “ \Rightarrow ” (Fig. 8). Rule (ax) generates a list of eight bytes in the `toCommit` component. The first one is the `write` term shown in the state, and the other seven bytes are represented by variable Δ . The variable B inside the `byte` construct is an eight bit list with all of 0 bits because we are getting the left-most eight bits of the [11,11] array. By applying rule (c), we get the resulting state after the second “ \Rightarrow ”. This rule moves the `write` operation from the component `toCommit` in thread φ to the empty component `memOpList` in the memory cache.

Next, rule (e) is executed and we get another new state after the third “ \Rightarrow ”. We can see that the components `race`, `complete`, and `byteMap` (Γ) are updated, and the `memOpList` component becomes empty. This process keeps going until all items in `toCommit` have reached `byteMap` (Γ).

Another example is group (t) in Figure 8. It represents the computations of the `store` instruction at line 22 of `Program-A` (Fig. 1). In the initial state, the pointer has the range attribute `none`, so the state is transitioned to an error state with the `unspecifiedBehavior` indicator.

Notice that in some states in Group (s), the system might have non-deterministic choices over transition rules. For these non-deterministic choices, we have the following important observation, which is clearly true in **K-LLVM** because the `toCommit` and `memOpList` components are in FIFO order, and each thread executes instructions in the program order in the `continuation` component.

► **Observation 2.** Assume that a trace of memory operations is generated by observing the order of memory operations committed to the `byteMap` in the memory cache. For a valid LLVM IR program, no matter which rule the **K-LLVM** abstract machine chooses to apply in a transition state if such rule correctly pattern matches the state, the memory trace generated by executing the program is byte-wise sequentially consistent.

The readonly Function Flag. LLVM IR allows users to set flags on the function headers that suggest that the function has certain features over memory instructions. The `readonly` flag is a representative. It means that the execution of the function with the flag should not use any memory write operations, e.g. a `store` instruction. If executing a function does use a write operation, it is unspecified behavior. In Figure 4, there is a `flags` component in the `control` component of a thread. During the static semantics step in Section 4.1, all functions from a LLVM IR program are compiled to BAST format and stored in a database, including function header flag information. During executing in the **K-LLVM** abstract machine, when a function is called, **K-LLVM** context switches the `control` component for the function, including the flag information called from the database and stored in the `flags` component. When **K-LLVM** is executing a `store` operation, according to the `store` rules (Fig. 7), **K-LLVM** checks if the `flags` contain a `readonly` flag. If not, the `store` operation can proceed; otherwise, the whole transition state is transitioned to an error state of `unspecifiedBehavior`.

We have given a general idea of how **K-LLVM** implements different semantic aspects of LLVM IR here. The full details from the real **K-LLVM** semantics in \mathbb{K} and another **K-LLVM** abstract machine in the Isabelle implementation [26].

5 Evaluation and Applications

Evaluating **K-LLVM** took more than half of the development time. We used \mathbb{K} to generate an interpreter for **K-LLVM** and ran LLVM IR programs in it. We mainly used the testing process as a tool to validate the correctness of our semantics, comprised of individual instruction semantics and our memory models. We also developed several tools to show the usage of **K-LLVM**.

Testing Process of K-LLVM. The validation of language semantics is usually accomplished through the use of external test suites [4, 11, 13], which was also part of our strategy. We use a large test suite to test the output of **K-LLVM** against Clang/Clang++. The tests are

split into two sets. We have a set of unit test cases containing totally 1,385 medium size test programs, and they were made in the process of defining K-LLVM. They are made to test each individual instruction or intrinsic function listed in the LLVM documentation with the consideration of all corner cases. We also have a set of regression test suite. There are 2,156 programs from the GCC-torture test suites. They are compiled from C to LLVM directly without optimizations. They are used as a regression test suite to validate K-LLVM. Besides, we also use the test suite (around 900 test cases) from previous K-Java semantics [5] as a regression test suite to test K-LLVM. We compiled the Java test cases from Java to LLVM without optimizations. For all of these cases, we first get the output from Clang/Clang++ for compiling a test program to machine code and executing it, and then compare the output with the output of executing the same program by **K-LLVM**. For validating the threading libraries (including mutex ones), we use the K state space exploration tool that will be introduced later in the section. In the unit test suite, we had 128 multi-threaded programs. We first execute them by the state space exploration tool, and get all traces (including all syscalls/memory operations) of each individual program, and examine manually if they are correct. The test cases and Clang bugs have been documented in the **K-LLVM** implementation [26], and the bugs have been reported to the LLVM community.

The methodology for developing **K-LLVM** was based on a strategy named Test Driven Development (TDD), whose basic idea is to develop tests before implementing the actual features. LLVM IR has an official test suite, but it is hard to break it down into individual pieces. In developing **K-LLVM**, the test principle is to test individual features while coordinating new features with old defined ones. When we defined a new feature in **K-LLVM**, we followed four steps. First, we read the details about the feature in the LLVM IR documentation, and thought about how to define the static and dynamic semantics of it. Next, we wrote out unit test cases to test the feature in the current LLVM IR implementation (Clang/Clang++). We made sure that we covered enough corner cases by designing a good set of new unit tests. We then defined the feature and tested it with the new unit tests, making sure it could pass them all. Third, we added the new feature to all of the defined unit tests to see if it caused any new problems. Finally, we tested the whole semantics with the regression test suite (the GCC-torture and K-Java test suites) and made sure that it passed more test cases than before and did not introduce new problems. When we developed **K-LLVM**, we started by defining the static semantics for each individual feature in LLVM IR, and made sure that all static features were validated for every variable, expression, instruction, function and module. After that, we defined the **K-LLVM** memory model and validated the correctness of the model. Following the definition of the model, we incrementally defined the semantics of the instructions, working from those that interacted least with other instructions and the memory such as the arithmetic and conversion instructions, through to the branching instructions and finally those that affected the memory. Lastly, we defined different memory operations. The distinction between the atomic and non-atomic memory operations is particularly complicated due to the fact that we define the non-atomic memory system to be based on reading/writing one byte at a time.

While searching for undesirable behaviors in Clang was not an objective of this project, we found some in the process of defining the **K-LLVM** semantics. Mainly, we ran test programs, and compared their outputs with those listed in the LLVM documentation. Undesirable behaviors happened in very diverse circumstances. A large number of them related to the fact that Clang does not place enough checks to validate what the LLVM IR documentation suggests. In other cases, Clang has missing features. For example, one cannot cast an `fp128` constant to a `ppc_fp128` constant, which should be allowed. In some cases, the description

of the LLVM documentation is not clear. For example, in describing the `fptrunc` and `fpext` instructions, LLVM IR uses the idea of large floating point types, and allows a comparison of two of them. However, it does not give a precise description of how to make this comparison. In fact, we found that the two types `fp128` and `ppc_fp128` are not comparable, so there is no way in LLVM IR to cast from one to the other, contrary to the documentation.

Finally, we use 128 multi-threaded programs to test the **K-LLVM** thread library with *ksearch*. **K-LLVM** produced a set of behaviors that are all expected according with respect to our thread and byte-wise sequentially consistent memory model. There are other multi-threaded programs used for testing the full memory concurrency behaviors, which is out-of-scope of the paper.

Morpheus on K-LLVM. We built the Morpheus tool [33] on top of **K-LLVM** to support correct specifications of compiler optimizations of LLVM IR programs. The Morpheus core language is a domain-specific one for formal specifications of program transformations. It describes program transformations as rewrites on control flow graphs with temporal logic (CTL) side conditions. Morpheus allows users to specify comprehensible program optimizations including those in data flow analysis and data dependence graph analysis. Its executable semantics allows these specifications to act as prototypes for the optimizations themselves, so that candidate optimizations can be tested and refined before including them in a compiler. We built Morpheus on top of **K-LLVM** in \mathbb{K} , so that users are able to specify program optimizations in LLVM IR, and test the optimizations by using \mathbb{K} tools for LLVM IR programs. Through the **IsaK** and **TransK** tools [28, 27], we translate **K-LLVM** into a transition system in Isabelle, and merge it with the Morpheus tool in Isabelle. With this system, we are able to prove the correctness of the optimizations in Isabelle under the assumption that programs are executed in the **K-LLVM** abstract machine and a choice of memory model. As an instance, we are able to define redundant store elimination properties on LLVM IR programs in Isabelle under sequential consistency. With the **K-LLVM** abstract machine, we have a framework for proving the correctness of the optimization for all programs in LLVM IR in Isabelle. The finalization of the proof will be an interesting future work of **K-LLVM**. The detailed semantics of Morpheus, and its union with a transition semantics for a fragment of LLVM for use in proving properties of program transformations is in [32], but **K-LLVM** came after the paper.

Detecting Undefined Behaviors. When an undefined behavior happens, **K-LLVM** outputs an error state. This is particularly useful for programmers to reveal unexpected behaviors to programmers, especially memory access errors. For example, in **Program-A** (Fig. 1), the execution of the program results in a transition state with an **error** component containing an `unspecifiedBehavior` construct (Fig. 8). This is because pointer `%r9` comes from a non-valid source. By using *krun*, we can see the following error message for the **Program-A** execution:

```
$ krun program-a.ll
ERROR while executing the program.
Description: The argument pointer points to an illegal location.
Line-number: 22
```

For some undefined behaviors in LLVM IR, the *ksearch* space exploration method cannot list all outputs. **Program-E** (Fig. 1) is such an example. The program is to create a memory field, get a memory pointer, then turn the pointer to an integer and print it. The output is a non-deterministic value with infinite many possible values. When using *krun* (the single-thread execution engine in \mathbb{K}) to execute the program, it prints out a random integer

value depending on the runtime memory address allocation in **K-LLVM**. A better way to analyze the program is to use the \mathbb{K} symbolic execution engine. One can use *ksearch* with the `-symbolic` flag to execute this program, and the final result is a variable representing an integer value. One can also use the \mathbb{K} symbolic equivalence checker to check if the executions of two similar programs printing out variables representing the same range of integers. The equivalence checker relies on the Z3 SMT solver to calculate if two variables representing the same range of values.

State Space Exploration. A trivial utility of **K-LLVM** is state space exploration through the *ksearch* tool. Users can use *ksearch* (actual command: `krun -search`) to see all possible final results and traces of multi-threaded programs based on the automatically generated interpreter for **K-LLVM** in \mathbb{K} . This can be useful for detecting out-of-thin-air behaviors. For example, by assuming sequential consistency, if we execute `program-B` (Fig. 1) with the initial values of zero in both memory fields for pointers `@x` and `@y`, the final results of `%a` and `%b` can never both be zero. We can also detect undefined values of a race. According to the documentation of LLVM IR, when a non-atomic `store` happens, and another memory operation from another thread is trying to access the same field, a race happens, and the two memory operations both get `undef`. By using *ksearch* to execute `program-C` (Fig. 1), we can see `undef` for variables `%a` and `%b` in some final results.

Additionally, the option `-pattern` allows us to filter the traces generated by executing a multi-threaded program. This option can be used to detect some interesting behaviors. For example, in **K-LLVM**, the `globalControl` component has a sub-component named `waitJoinThreads` that is used to store the states when a thread is waiting to join its child threads. If two threads in **K-LLVM** use the Pthread library function `pthread_join` to wait for each other in a multi-threaded program, the result is a deadlock. We can use the `-pattern` option with the pattern $\langle M (X \mid \rightarrow \text{EDEADLK}) \rangle_{\text{waitJoinThreads}}$ to detect if any trace of the multi-threaded program results in a deadlock. The key word `EDEADLK` is a flag in the Pthread library meaning that a thread has ended in a deadlock. Variable `X` represents any thread with an unspecified thread ID.

6 Conclusion and Future Work

In this paper, we propose **K-LLVM**, a formal semantics of LLVM IR in \mathbb{K} . The main advantages of **K-LLVM** is its relatively completeness and its implementation via a novel abstract machine for LLVM IR. To the best of our knowledge, **K-LLVM** is the most complete formal semantics of LLVM IR. We fully define the static semantics and dynamic semantics of LLVM IR relative to a sequentially consistent memory model. To validate its completeness, we ran 1,385 unit testing and around 3,000 concrete test programs, all of which **K-LLVM** successfully executed. **K-LLVM** provides guidance and reference to future compiler developers on exactly what are permissible behaviors in running LLVM IR programs. It also provides important piece of a framework for proving properties of compilers or from LLVM IR. The **K-LLVM** abstract machine is a concise way of specifying how each LLVM IR instruction interacts with different computer components. In particular, **K-LLVM** covers corner cases and side-effects of instruction semantics that previous work does not have, such as the different cases of the `getelementptr` operators, casting operators, and memory operators. **K-LLVM** also supports multi-threaded behaviors and provides users a collection of tools, including a state-space searching tool to explore traces of their LLVM IR programs under the assumption of sequential consistency. While this was not the main focus of this work, we also found more than 20 bugs in the current LLVM implementation, Clang.

In follow-on work to this paper, we have two on-going studies of **K-LLVM**. First, we are trying to finalize the full LLVM IR memory model in **K-LLVM**, including the behaviors of different atomic memory orderings and `volatile` memory accesses, with heavy testings and proofs of its relationship with existing C++ memory models [1, 19, 20, 16, 40, 8]. Second, we are defining a formal semantics for Haskell and verifying the correctness of the compiler from Haskell to LLVM IR, which requires both the semantics of Haskell and the semantics of LLVM IR as given in this paper.

References

- 1 Mark Batty, Scott Owens, Susmit Sarkar, Peter Sewell, and Tjark Weber. Mathematizing C++ Concurrency. *SIGPLAN Not.*, 46(1):55–66, January 2011. doi:10.1145/1925844.1926394.
- 2 Yves Bertot and Pierre Casteran. *Interactive Theorem Proving and Program Development*. SpringerVerlag, 2004.
- 3 Sandrine Blazy and Xavier Leroy. Mechanized Semantics for the Clight Subset of the C Language. *Journal of Automated Reasoning*, 43(3):263–288, 2009. doi:10.1007/s10817-009-9148-3.
- 4 Martin Bodin, Arthur Chargueraud, Daniele Filaretti, Philippa Gardner, Sergio Maffei, Daiva Naudziuniene, Alan Schmitt, and Gareth Smith. A Trusted Mechanised JavaScript Specification. *SIGPLAN Not.*, 49(1):87–100, January 2014. doi:10.1145/2578855.2535876.
- 5 Denis Bogdănaş and Grigore Roşu. K-Java: A Complete Semantics of Java. In *Proceedings of the 42nd Symposium on Principles of Programming Languages (POPL'15)*, pages 445–456. ACM, January 2015. doi:10.1145/2676726.2676982.
- 6 Egon Börger, Nicu G. Fruja, Vincenzo Gervasi, and Robert F. Stärk. A High-level Modular Definition of the Semantics of C#. *Theor. Comput. Sci.*, 336(2-3):235–284, May 2005. doi:10.1016/j.tcs.2004.11.008.
- 7 Soham Chakraborty and Viktor Vafeiadis. Formalizing the Concurrency Semantics of an LLVM Fragment. In *Proceedings of the 2017 International Symposium on Code Generation and Optimization, CGO '17*, pages 100–110, Piscataway, NJ, USA, 2017. IEEE Press. URL: <http://dl.acm.org/citation.cfm?id=3049832.3049844>.
- 8 Soham Chakraborty and Viktor Vafeiadis. Grounding Thin-air Reads with Event Structures. *Proc. ACM Program. Lang.*, 3(POPL):70:1–70:28, January 2019. doi:10.1145/3290383.
- 9 Sophia Drossopoulou, Susan Eisenbach, and Sarfraz Khurshid. Is the Java Type System Sound? *Theor. Pract. Object Syst.*, 5(1):3–24, January 1999. doi:10.1002/(SICI)1096-9942(199901/03)5:1<3::AID-TAP02>3.0.CO;2-T.
- 10 Chucky Ellison and David Lazar. The LLVM Semantics in K, 2012. URL: <https://github.com/davidlazar/llvm-semantics>.
- 11 Chucky Ellison and Grigore Rosu. An Executable Formal Semantics of C with Applications. In *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '12, pages 533–544, New York, NY, USA, 2012. ACM. doi:10.1145/2103656.2103719.
- 12 Azadeh Farzan, Feng Chen, José Meseguer, and Grigore Roşu. Formal Analysis of Java Programs in JavaFAN. In Rajeev Alur and Doron A. Peled, editors, *Computer Aided Verification*, pages 501–505, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.
- 13 Daniele Filaretti and Sergio Maffei. An Executable Formal Semantics of PHP. In Richard Jones, editor, *ECOOP 2014 – Object-Oriented Programming*, pages 567–592, Berlin, Heidelberg, 2014. Springer Berlin Heidelberg.
- 14 Yuri Gurevich. Evolving algebras 1993: Lipari guide. In Egon Börger, editor, *Specification and Validation Methods*, pages 9–36. Oxford University Press, Inc., New York, NY, USA, 1995. URL: <http://dl.acm.org/citation.cfm?id=233976.233979>.

- 15 Myra Van Inwegen and Elsa L. Gunter. HOL-ML. In *Proceedings of the 6th International Workshop on Higher Order Logic Theorem Proving and Its Applications*, HUG '93, pages 61–74, London, UK, UK, 1994. Springer-Verlag. URL: <http://dl.acm.org/citation.cfm?id=646520.694367>.
- 16 Jeehoon Kang, Chung-Kil Hur, Ori Lahav, Viktor Vafeiadis, and Derek Dreyer. A Promising Semantics for Relaxed-memory Concurrency. *SIGPLAN Not.*, 52(1):175–189, January 2017. doi:10.1145/3093333.3009850.
- 17 Jeehoon Kang, Chung-Kil Hur, William Mansky, Dmitri Garbuzov, Steve Zdancewic, and Viktor Vafeiadis. A Formal C Memory Model Supporting Integer-pointer Casts. *SIGPLAN Not.*, 50(6):326–335, June 2015. doi:10.1145/2813885.2738005.
- 18 Jeehoon Kang, Yoonseung Kim, Youngju Song, Juneyoung Lee, Sanghoon Park, Mark Dongyeon Shin, Yonghyun Kim, Sungkeun Cho, Joonwon Choi, Chung-Kil Hur, and Kwangkeun Yi. Crellvm: Verified Credible Compilation for LLVM. *SIGPLAN Not.*, 53(4):631–645, June 2018. doi:10.1145/3296979.3192377.
- 19 Ori Lahav, Nick Giannarakis, and Viktor Vafeiadis. Taming Release-acquire Consistency. *SIGPLAN Not.*, 51(1):649–662, January 2016. doi:10.1145/2914770.2837643.
- 20 Ori Lahav, Viktor Vafeiadis, Jeehoon Kang, Chung-Kil Hur, and Derek Dreyer. Repairing Sequential Consistency in C/C++11. *SIGPLAN Not.*, 52(6):618–632, June 2017. doi:10.1145/3140587.3062352.
- 21 Chris Lattner and Vikram Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization*, CGO '04, pages 75–, Washington, DC, USA, 2004. IEEE Computer Society. URL: <http://dl.acm.org/citation.cfm?id=977395.977673>.
- 22 Daniel K. Lee, Karl Cray, and Robert Harper. Towards a Mechanized Metatheory of Standard ML. *SIGPLAN Not.*, 42(1):173–184, January 2007. doi:10.1145/1190215.1190245.
- 23 Juneyoung Lee, Chung-Kil Hur, Ralf Jung, Zhengyang Liu, John Regehr, and Nuno P. Lopes. Reconciling High-level Optimizations and Low-level Code in LLVM. *Proc. ACM Program. Lang.*, 2(OOPSLA):125:1–125:28, October 2018. doi:10.1145/3276495.
- 24 Juneyoung Lee, Yoonseung Kim, Youngju Song, Chung-Kil Hur, Sanjoy Das, David Majnemer, John Regehr, and Nuno P. Lopes. Taming Undefined Behavior in LLVM. *SIGPLAN Not.*, 52(6):633–647, June 2017. doi:10.1145/3140587.3062343.
- 25 Xavier Leroy, Andrew W. Appel, Sandrine Blazy, and Gordon Stewart. The CompCert Memory Model, Version 2. Research Report RR-7987, INRIA, June 2012. URL: <https://hal.inria.fr/hal-00703441>.
- 26 Liyi Li and Elsa Gunter. LLVM Semantics, 2019. URL: <https://github.com/liyili2/llvm-semantics-1>.
- 27 Liyi Li and Elsa L. Gunter. IsaK: A Complete Semantics of K. Technical Report <http://hdl.handle.net/2142/100116>, University of Illinois at Urbana-Champaign, June 2018.
- 28 Liyi Li and Elsa L. Gunter. IsaK-Static: A Complete Static Semantics of K. In Kyungmin Bae and Peter Csaba Ölveczky, editors, *Formal Aspects of Component Software*, pages 196–215, Cham, 2018. Springer International Publishing.
- 29 llvm.org. LLVM Language Reference Manual, 2018. URL: <http://releases.llvm.org/6.0.0/docs/LangRef.html>.
- 30 Nuno P. Lopes, David Menendez, Santosh Nagarakatte, and John Regehr. Provably Correct Peephole Optimizations with Alive. *SIGPLAN Not.*, 50(6):22–32, June 2015. doi:10.1145/2813885.2737965.
- 31 Savi Maharaj and Elsa L. Gunter. Studying the ML Module System in HOL. In *Proceedings of the 7th International Workshop on Higher Order Logic Theorem Proving and Its Applications*, pages 346–361, London, UK, UK, 1994. Springer-Verlag. URL: <http://dl.acm.org/citation.cfm?id=646521.759249>.

- 32 William Mansky. Specifying and verifying program transformations with PTRANS. Technical Report <http://hdl.handle.net/2142/49385>, University of Illinois at Urbana-Champaign, May 2014.
- 33 William Mansky, Elsa L. Gunter, Dennis Griffith, and Michael D. Adams. Specifying and Executing Optimizations for Generalized Control Flow Graphs. *Science of Computer Programming*, 130:2–23, November 2016. doi:10.1016/j.scico.2016.06.003.
- 34 Narciso Martí-Oliet and José Meseguer. Rewriting Logic as a Logical and Semantic Framework. In J. Meseguer, editor, *Electronic Notes in Theoretical Computer Science*, volume 4. Elsevier Science Publishers, 2000.
- 35 Paul E. Mckenney. Memory Barriers: a Hardware View for Software Hackers, 2009.
- 36 Kayvan Memarian, Victor B. F. Gomes, Brooks Davis, Stephen Kell, Alexander Richardson, Robert N. M. Watson, and Peter Sewell. Exploring C Semantics and Pointer Provenance. *Proc. ACM Program. Lang.*, 3(POPL):67:1–67:32, January 2019. doi:10.1145/3290380.
- 37 Robin Milner, Mads Tofte, and David MacQueen. *The Definition of Standard ML*. MIT Press, Cambridge, MA, USA, 1997.
- 38 Daejun Park, Andrei Ştefănescu, and Grigore Roşu. KJS: A Complete Formal Semantics of JavaScript. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'15)*, pages 346–356. ACM, June 2015. doi:10.1145/2737924.2737991.
- 39 Lawrence C. Paulson. Isabelle: The Next 700 Theorem Provers. In P. Odifreddi, editor, *Logic and Computer Science*, pages 361–386. Academic Press, 1990.
- 40 Anton Podkopaev, Ori Lahav, and Viktor Vafeiadis. Bridging the gap between programming languages and hardware weak memory models. *Proc. ACM Program. Lang.*, 3(POPL):69:1–69:31, January 2019. doi:10.1145/3290382.
- 41 Grigore Roşu. K Implementation, 2016. URL: <https://github.com/kframework/k>.
- 42 Grigore Roşu and Traian Florin Şerbănuţă. An Overview of the K Semantic Framework. *Journal of Logic and Algebraic Programming*, 79(6):397–434, 2010. doi:10.1016/j.jlap.2010.03.012.
- 43 Don Syme. Proving Java Type Soundness. In *Formal Syntax and Semantics of Java*, pages 83–118, London, UK, UK, 1999. Springer-Verlag. URL: <http://dl.acm.org/citation.cfm?id=645580.658814>.
- 44 Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. Finding and Understanding Bugs in C Compilers. *SIGPLAN Not.*, 46(6):283–294, June 2011. doi:10.1145/1993316.1993532.
- 45 Jianzhou Zhao, Santosh Nagarakatte, Milo M.K. Martin, and Steve Zdancewic. Formalizing the LLVM Intermediate Representation for Verified Program Transformations. *SIGPLAN Not.*, 47(1):427–440, January 2012. doi:10.1145/2103621.2103709.


Revision Notice

This is a revised version of the eponymous paper appeared in the proceedings of ECOOP 2020 (LIPIcs, volume 166, <http://www.dagstuhl.de/dagpub/978-3-95977-154-2>, published in November, 2020), in which the following changes were made:

- Page 4, Section 2, the following sentence is added: “The K-LLVM semantics builds on top of the LLVM semantics in K by Ellison and Lazar [10]. Our semantics directly extends their work to support missing features, including a more precise memory model and concurrency”.
- Page 4, the first paragraph is replaced with: “Other LLVM IR Semantics. Besides K-LLVM, the other formal executable semantics we keep track of the provenance information for the lifetime of the pointer no matter what the pointer becomes. The details are in Sec. 4.4.”.
- Page 27, the reference [10] by Chucky Ellison and David Lazar is added.

Dagstuhl Publishing – April 12, 2021.

Space-Efficient Gradual Typing in Coercion-Passing Style

Yuya Tsuda 

Graduate School of Informatics, Kyoto University, Japan
tsuda@fos.kuis.kyoto-u.ac.jp

Atsushi Igarashi 

Graduate School of Informatics, Kyoto University, Japan
igarashi@kuis.kyoto-u.ac.jp

Tomoya Tabuchi

Graduate School of Informatics, Kyoto University, Japan
tabuchi@fos.kuis.kyoto-u.ac.jp

Abstract

Herman et al. pointed out that the insertion of run-time checks into a gradually typed program could hamper tail-call optimization and, as a result, worsen the space complexity of the program. To address the problem, they proposed a space-efficient coercion calculus, which was subsequently improved by Siek et al. The semantics of these calculi involves eager composition of run-time checks expressed by coercions to prevent the size of a term from growing. However, it relies also on a nonstandard reduction rule, which does not seem easy to implement. In fact, no compiler implementation of gradually typed languages fully supports the space-efficient semantics faithfully.

In this paper, we study *coercion-passing style*, which Herman et al. have already mentioned, as a technique for straightforward space-efficient implementation of gradually typed languages. A program in coercion-passing style passes “the rest of the run-time checks” around – just like continuation-passing style (CPS), in which “the rest of the computation” is passed around – and (unlike CPS) composes coercions eagerly. We give a formal coercion-passing translation from λS by Siek et al. to λS_1 , which is a new calculus of *first-class coercions* tailored for coercion-passing style, and prove correctness of the translation. We also implement our coercion-passing style transformation for the Grift compiler developed by Kuhlenschmidt et al. An experimental result shows stack overflow can be prevented properly at the cost of up to 3 times slower execution for most partially typed practical programs.

2012 ACM Subject Classification Theory of computation → Semantics and reasoning; Software and its engineering → Compilers; Theory of computation → Operational semantics

Keywords and phrases Gradual typing, coercion calculus, coercion-passing style, dynamic type checking, tail-call optimization

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2020.8

Related Version A full version of the paper is available at <https://arxiv.org/abs/1908.02414>.

Funding This work was partially supported by JSPS KAKENHI Grant Number JP17H01723.

Acknowledgements We thank anonymous reviewers for valuable comments and John Toman for proofreading.

1 Introduction

1.1 Space-Efficiency Problem in Gradual Typing

Gradual typing [36, 40] is one of the linguistic approaches to integrating static and dynamic typing. Allowing programmers to mix statically typed and dynamically typed fragments in a single program, it advocates the “script to program” evolution [40]. Namely, software



© Yuya Tsuda, Atsushi Igarashi, and Tomoya Tabuchi;
licensed under Creative Commons License CC-BY

34th European Conference on Object-Oriented Programming (ECOOP 2020).

Editors: Robert Hirschfeld and Tobias Pape; Article No. 8; pp. 8:1–8:29

Leibniz International Proceedings in Informatics



LIPICs Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

8:2 Space-Efficient Gradual Typing in Coercion-Passing Style

development starts with simple, often dynamically typed scripts, which evolve to more robust, fully statically typed programs through intermediate stages of partially typed programs. To make this evolution work in practice, it is important that the performance of partially typed programs at intermediate stages is comparable to that of (the slower of) the two ends, that is, dynamically typed scripts and statically typed programs.

However, it has been pointed out that gradual typing suffers from serious efficiency problems from both theoretical and practical viewpoints [19, 20, 39]. In particular, Takikawa et al. [39] showed that even a state-of-the-art gradual typing implementation could show catastrophic slowdown for partially typed programs due to run-time checking to ensure safety. Worse, such slowdown is not easy to predict because it depends on implicit run-time checks inserted by the language implementation and it requires fairly deep knowledge about the underlying gradual type system to understand when and where run-time checks are inserted and how they behave. Since then, several pieces of work have investigated the performance issues [4, 27, 31, 29, 24, 12].

Earlier work by Herman et al. [19, 20] pointed out a related problem. They showed that, when values are passed between a statically typed part and a dynamically typed part many times, delayed run-time checks may accumulate and make space complexity of a program worse than an unchecked semantics.

To make the discussion more concrete, consider the following mutually recursive functions (written in ML-like syntax):

```
let rec even (x : int) : * =
  if x = 0 then true⟨bool!⟩ else (odd (x - 1))⟨bool!⟩
and odd (x : int) : bool =
  if x = 0 then false else (even (x - 1))⟨bool?p⟩
```

Ignoring the gray part (in angle brackets), which will be explained shortly, this is a tail-recursive definition of functions to decide whether a given integer is even or odd, except that the return type of one of the functions is written \star , which is the dynamic type, which can be any tagged value. This definition expresses a situation where a statically typed and a dynamically typed function call each other.¹ The gray part represents inserted run-time checks, written using Henglein’s coercion syntax [18]: bool! is a coercion from bool to \star and $\text{true}\langle\text{bool!}\rangle$ means that (untagged) Boolean value true will be tagged with bool to make a value of the dynamic type; bool?^p is a coercion from \star to bool and $(\text{even } (x - 1))\langle\text{bool?}^p\rangle$ means that the value returned from recursive call $\text{even } (x - 1)$ will be tested whether it is tagged with bool – if so, the run-time check removes the tag and returns the untagged Boolean value, and, otherwise, it results in *blame*, which is an uncatchable exception (with label p to indicate where the check has failed).

The crux of this example is that the insertion of run-time checks has broken tail recursion: due to $\langle\text{bool!}\rangle$ and $\langle\text{bool?}^p\rangle$, the recursive calls are not in tail positions any longer. So, according to the original semantics of coercions [18], evaluation of $\text{odd } 4$ is as follows:

$$\begin{aligned} \text{odd } 4 &\mapsto^* (\text{even } 3)\langle\text{bool?}^p\rangle \mapsto^* (\text{odd } 2)\langle\text{bool!}\rangle\langle\text{bool?}^p\rangle \\ &\mapsto^* (\text{even } 1)\langle\text{bool?}^p\rangle\langle\text{bool!}\rangle\langle\text{bool?}^p\rangle \mapsto^* (\text{odd } 0)\langle\text{bool!}\rangle\langle\text{bool?}^p\rangle\langle\text{bool!}\rangle\langle\text{bool?}^p\rangle \\ &\mapsto^* \text{false}\langle\text{bool!}\rangle\langle\text{bool?}^p\rangle\langle\text{bool!}\rangle\langle\text{bool?}^p\rangle \mapsto^* \text{false} \end{aligned}$$

¹ In this sense, the argument of even should have been \star , too, but it would clutter the code after inserting run-time checks.

$ \begin{aligned} &\text{odd } 4 \\ &\mapsto^* (\text{even } 3)\langle \text{bool}^{?^p} \rangle \\ &\mapsto (\text{odd } (3 - 1))\langle \text{bool}! \rangle\langle \text{bool}^{?^p} \rangle \\ &\mapsto (\text{odd } (3 - 1))\langle \text{bool}! \ ; \ \text{bool}^{?^p} \rangle \\ &= (\text{odd } (3 - 1))\langle \text{id}_{\text{bool}} \rangle \\ &\mapsto (\text{odd } 2)\langle \text{id}_{\text{bool}} \rangle \\ &\mapsto (\text{even } (2 - 1))\langle \text{bool}^{?^p} \rangle\langle \text{id}_{\text{bool}} \rangle \\ &\mapsto (\text{even } (2 - 1))\langle \text{bool}^{?^p} \ ; \ \text{id}_{\text{bool}} \rangle \\ &= (\text{even } (2 - 1))\langle \text{bool}^{?^p} \rangle \\ &\mapsto (\text{even } 1)\langle \text{bool}^{?^p} \rangle \\ &\mapsto \dots \end{aligned} $	$ \begin{aligned} &\text{oddk } (4, \text{id}_{\text{bool}}) \\ &\mapsto \text{evenk } (4 - 1, \text{bool}^{?^p} \ ; \ ; \ \text{id}_{\text{bool}}) \\ &\mapsto \text{evenk } (4 - 1, \text{bool}^{?^p}) \\ &\mapsto \text{evenk } (3, \text{bool}^{?^p}) \\ &\mapsto \text{oddk } (3 - 1, \text{bool}! \ ; \ ; \ \text{bool}^{?^p}) \\ &\mapsto \text{oddk } (3 - 1, \text{id}_{\text{bool}}) \\ &\mapsto \text{oddk } (2, \text{id}_{\text{bool}}) \\ &\mapsto \text{evenk } (2 - 1, \text{bool}^{?^p} \ ; \ ; \ \text{id}_{\text{bool}}) \\ &\mapsto \text{evenk } (2 - 1, \text{bool}^{?^p}) \\ &\mapsto \text{evenk } (1, \text{bool}^{?^p}) \\ &\mapsto \dots \end{aligned} $
--	---

■ **Figure 1** Reduction from $\text{odd } 4$ in $\lambda\mathcal{S}$ (left) and reduction from $\text{oddk } (4, \text{id}_{\text{bool}})$ in $\lambda\mathcal{S}_1$ (right).

Thus, the size of a term being evaluated is proportional to the argument n at its longest, whereas unchecked semantics (without coercions) allows for tail-call optimization and constant-space execution. This is the space-efficiency problem of gradual typing.

1.2 Space-Efficient Gradual Typing

Herman et al. [19, 20] also presented a solution to this problem. In the evaluation sequence of $\text{odd } n$ above, we could immediately “compress” nested coercion applications $M\langle \text{bool}! \rangle\langle \text{bool}^{?^p} \rangle$ before computation of the target term M ends, because $\langle \text{bool}! \rangle\langle \text{bool}^{?^p} \rangle$ – tagging immediately followed by untagging – is equivalent to the identity function. By doing so, we can maintain that the order of the size of a term in the middle of evaluation is constant. This idea is formalized in terms of a “space-efficient” extension of the coercion calculus [18]. Since then, a few space-efficient coercion/cast calculi have been proposed [37, 38, 35].

Among them, Siek et al. [37] have proposed a space-efficient coercion calculus $\lambda\mathcal{S}$. $\lambda\mathcal{S}$ is equipped with a composition function that compresses consecutive coercions in certain canonical forms. The coercion composition is achieved as a simple recursive function thanks to the canonical forms. We show evaluation of $\text{odd } 4$ according to the $\lambda\mathcal{S}$ semantics in the left of Figure 1.² Here, $s \ ; \ t$ is a meta-level operation that composes two coercions s, t (in canonical forms) and yields another canonical coercion that semantically corresponds to their sequential composition. This composition function enables us to prevent the size of a term from growing.

However, in order to ensure that nested coercion applications are always merged, the operational semantics of $\lambda\mathcal{S}$ relies on a nonstandard reduction rule and nonstandard evaluation contexts. Although it does not cause any theoretical problems, it does not seem easy to implement – in particular, its compilation method seems nontrivial. In fact, none of the existing compiler implementations that address the space-efficiency problem [24, 12] solves the problem of growing coercions at tail positions (an exception is recent work by Castagna et al. [5] – See Section 6 for more comparison).

² Strictly speaking, $\text{bool}!$ and $\text{bool}^{?^p}$ are abbreviations of $\text{id}_{\text{bool}}; \text{bool}!$ and $\text{bool}^{?^p}; \text{id}_{\text{bool}}$, respectively, in $\lambda\mathcal{S}$.

1.3 Our Work: Coercion-Passing Style

In this paper, we study coercion-passing style for space-efficient gradual typing. Just as continuation-passing style, in which “the rest of the computation” is passed around as first-class functions and every function call is at a tail position, a program in coercion-passing style passes “the rest of the run-time checks” around. Actually, the idea of coercion-passing style has already been listed as one of the possible implementation techniques by Herman et al. [19, 20] but it has been neither well studied nor formalized.

We use the even/odd example above to describe our approach to the problem. Here are the even/odd functions in coercion-passing style. (We omit type declarations for simplicity.)

```
let rec evenk (x, κ) =
  if x = 0 then true⟨bool! ;; κ⟩ else oddk (x - 1, bool! ;; κ)
and oddk (x, κ) =
  if x = 0 then false⟨κ⟩ else evenk (x - 1, bool?p ;; κ)
```

Additional parameters named κ are for *first-class coercions*, which are supposed to be applied – as in $\text{false}\langle\kappa\rangle$ – to values that are returned in the original function definition. We often call these coercions *continuation coercions*. Coercion applications such as $\text{true}\langle\text{bool!}\rangle$ and $(\text{oddk } (x - 1))\langle\text{bool!}\rangle$ at tail positions in the original program are translated to coercion compositions such as $\text{true}\langle\text{bool!} ;; \kappa\rangle$ and $\text{oddk } (x - 1, \text{bool!} ;; \kappa)$, respectively. When κ is bound to a concrete coercion, it will be composed with bool! *before it is applied*. Similarly to programs in CPS, function calls pass (composed) coercions.

With these functions in coercion-passing style, the evaluation of $\text{oddk } (4, \text{id}_{\text{bool}})$ (where id_{bool} is an identity coercion, which does nothing) proceeds as in the right of Figure 1. Since tagging followed by untagging (with the same tag) actually does nothing, $\text{bool!} ;; \text{bool?}^p$ composes to id_{bool} by the (meta-level) coercion composition $\text{bool!} \S \text{bool?}^p$.

Similarly to the $\lambda\mathcal{S}$ semantics described above, coercion composition in the argument takes place before a recursive call, thus the size of coercions stays bounded by the constant order, overcoming the space efficiency problem. A nice property of our solution is that the evaluation is standard call-by-value.

One can view the extra parameter κ as an accumulating parameter and continuation coercions as (delimited) continuations in defunctionalized forms [30]. Unlike simple defunctionalization, however, special composition of two defunctionalized coercions is provided, preventing the sizes of composed coercions from growing.

Contributions

Since the operational semantics of $\lambda\mathcal{S}$ seems nontrivial to implement due to a nonstandard reduction rule, we investigate implementation of the space-efficient semantics via a translation into coercion-passing style. Our contributions in this paper are summarized as follows:

- In the context of the space-efficiency problem of gradual typing, we develop a new calculus $\lambda\mathcal{S}_1$ of space-efficient first-class coercions.
- We formalize a coercion-passing style translation from (a slight variant of) space-efficient coercion calculus $\lambda\mathcal{S}$ [37] to the new calculus $\lambda\mathcal{S}_1$.
- We prove correctness of the coercion-passing style translation via a simulation property.
- We implement the coercion-passing style translation on top of the Grift compiler [24], and conduct some experiments to show that stack overflow is indeed avoided.

Outline

The rest of this paper is organized as follows. We review the space-efficient coercion calculus $\lambda\mathcal{S}$ [37] in Section 2. We introduce a new space-efficient coercion calculus with first-class coercions $\lambda\mathcal{S}_1$ in Section 3, formalize a translation into coercion-passing style as a translation from $\lambda\mathcal{S}$ to $\lambda\mathcal{S}_1$, and prove correctness of the translation in Section 4. We discuss our implementation of coercion-passing translation on top of the Grift compiler [24] and show an experimental result in Section 5. Finally, we discuss related work in Section 6 and conclude in Section 7. Proofs of the stated properties can be found in the full version.

2 Space-Efficient Coercion Calculus

In this section, we review the space-efficient coercion calculus $\lambda\mathcal{S}$ [37], which is the source calculus of our translation. Our definition differs from the original in a few respects, as we will explain later. For simplicity, we do not include (mutually) recursive functions and conditional expressions in the formalization but it is straightforward to add them; in fact, our implementation includes them.

Main novelties of $\lambda\mathcal{S}$ over the original coercion calculus $\lambda\mathcal{C}$ [18] are (1) space-efficient coercions, which are canonical forms of coercions, whose composition can be defined by a straightforward recursive function, and (2) operational semantics in which a sequence of coercion applications is collapsed eagerly – even before they are applied to a value [19, 20, 35].

Basic forms of coercions are inherited from $\lambda\mathcal{C}$ [18], which provides (1) identity coercions id_A (where A is a type), which do nothing; (2) injections $G!$, which add a type tag G to a value to make a value of the dynamic type; (3) projections $G?^p$, which test whether a value of the dynamic type is tagged with G , remove the tag if the test succeeds, or raise blame labeled p if it fails; (4) function coercions $c_1 \rightarrow c_2$, which, when they are applied to a function, coerce an argument to the function by c_1 and a value returned from the function by c_2 ; and (5) sequential compositions $c_1; c_2$, which apply c_1 and c_2 in this order. Space-efficient coercions restrict the way basic coercions are combined by sequential composition; they can be roughly expressed by the following regular expression:

$$(G?^p;)^?(\text{id}_\iota + (s_1 \rightarrow s_2)); G!^?)^?$$

(where ι is a base type, s_1 and s_2 stand for space efficient coercions, $(\dots)^?$ stands for an optional element, and $+$ for alternatives). As already mentioned, an advantage of this form is that (meta-level) sequential composition (denoted by $s_1 \mathbin{\text{;}} s_2$) of two space-efficient coercions results in another space-efficient coercion (if the composition is well typed), in other words, space-efficient coercions are closed under $s_1 \mathbin{\text{;}} s_2$. For example, the composition $((G_1?^p;)^?(\text{id}_\iota + (s_1 \rightarrow s_2)); G_2!) \mathbin{\text{;}} (G_3?^{p'}; (\text{id}_\iota + (s_3 \rightarrow s_4)); G_4!)^?$ will be $((G_1?^p;)^?(\text{id}_\iota + ((s_3 \mathbin{\text{;}} s_1) \rightarrow (s_2 \mathbin{\text{;}} s_4))); G_4!)^?$ if $G_2 = G_3$ – that is, tagging with G_2 is immediately followed by inspection whether G_2 is present.³ Notice that the resulting coercion conforms to the regular expression again. (The other case where $G_2 \neq G_3$ means that the projection $G_3?^{p'}$ will fail; we will explain such failures later.)

The operational semantics includes the reduction rule $\mathcal{F}[M\langle s \rangle\langle t \rangle] \longrightarrow \mathcal{F}[M\langle s \mathbin{\text{;}} t \rangle]$ where \mathcal{F} is an evaluation context that does not include nested coercion applications and whose innermost frame is not a coercion application. This rule intuitively means that two consecutive coercions at the outermost position will be composed *even before M is evaluated to a value*. This eager composition avoids a long chain of coercion applications in an evaluation context.

³ Here, we exclude ill-typed coercion compositions such as $(s_1 \rightarrow s_2) \mathbin{\text{;}} \text{id}_\iota$.

Variables	x, y	Constants	a, b	Operators	op	Blame labels	p
Base types			$\iota ::= \text{int} \mid \text{bool} \mid \dots$				
Types			$A, B, C ::= \star \mid \iota \mid A \rightarrow B$				
Ground types			$G, H ::= \iota \mid \star \rightarrow \star$				
Space-efficient coercions			$s, t ::= \text{id}_\star \mid G^{?P}; i \mid i$				
Intermediate coercions			$i ::= g; G! \mid g \mid \perp^{GpH}$				
Ground coercions			$g, h ::= \text{id}_A \text{ (if } A \neq \star) \mid s \rightarrow t \text{ (if } s \neq \text{id or } t \neq \text{id)}$				
Delayed coercions			$d ::= g; G! \mid s \rightarrow t \text{ (if } s \neq \text{id or } t \neq \text{id)}$				
Terms			$L, M, N ::= V \mid op(M, N) \mid MN \mid M\langle s \rangle \mid \text{blame } p$				
Values			$V, W ::= x \mid U \mid U\langle\langle d \rangle\rangle$				
Uncoerced values			$U ::= a \mid \lambda x. M$				
Type environments			$\Gamma ::= \emptyset \mid \Gamma, x : A$				

■ **Figure 2** Syntax of $\lambda\mathcal{S}$.

2.1 Syntax

We show the syntax of $\lambda\mathcal{S}$ in Figure 2. The syntax of $\lambda\mathcal{S}$ extends that of the simply typed lambda calculus (written in gray) with the dynamic type and (space-efficient) coercions.

Types, ranged over by A, B, C , include the dynamic type \star , base types ι , and function types $A \rightarrow B$. Base types ι include `int` (integer type) and `bool` (Boolean type) and so on. *Ground types*, ranged over by G, H , include base types ι and the function type $\star \rightarrow \star$. They are used for type tags put on values of the dynamic type [43]. Here, the ground type for functions is always $\star \rightarrow \star$, reflecting the fact that many dynamically typed languages do not include information on the argument and return types of the function in its type tag.

As we have already discussed, $\lambda\mathcal{S}$ restricts coercions to only canonical ones, namely space-efficient coercions s , whose grammar is defined via ground coercions g and intermediate coercions i . Ground coercions correspond to the middle part of space-efficient coercions; unlike the original $\lambda\mathcal{S}$, ground coercions include identity coercions for any function types – such as $\text{id}_{\iota \rightarrow \iota}$ – and exclude “virtually identity” coercions such as $\text{id}_{\iota} \rightarrow \text{id}_{\iota}$. Although these two coercions are extensionally the same, they reduce in slightly different ways: applying $\text{id}_{\iota \rightarrow \iota}$ to a function immediately returns the function, whereas applying $\text{id}_{\iota} \rightarrow \text{id}_{\iota}$ results in a wrapped function whose argument and return values are monitored by id_{ι} , which does nothing. Adopting id_A for any A simplifies our proof that the coercion-passing translation preserves the semantics. An intermediate coercion adds an optional injection to a ground coercion. Coercions of the form \perp^{GpH} trigger blame (labeled p) if applied to a value. They emerge from coercion composition

$$((G_1^{?P};)^?(\text{id}_A + (s_1 \rightarrow s_2)); G_2!) \circ (G_3^{?P'}; (\text{id}_A + (s_3 \rightarrow s_4)))(; G_4!)^?$$

where $A \neq \star$ and $G_2 \neq G_3$, which means that the projection $G_3^{?P'}$ is bound to fail. The composition results in $(G_1^{?P};)^? \perp^{G_1 P' G_3}$, which means that, unless the optional projection fails – blaming p – it fails with p' . Finally, space-efficient coercions are obtained by adding optional projection to intermediate coercions. id_\star is a special coercion that does not conform to the regular expression above. Strictly speaking, an injection, say `int!`, has to be written $\text{id}_{\text{int}}; \text{int!}$ and a projection, say `int?P`, has to be written $\text{int}^{?P}; \text{id}_{\text{int}}$. We often omit these identity coercions in examples.

Terms, ranged over by L, M, N , include values V , primitive binary operations $op(M, N)$, function applications $M N$, coercion applications $M\langle s \rangle$, and coercion failure $\mathbf{blame} p$. The term $M\langle s \rangle$ coerces the value of M with coercion s at run time. The term $\mathbf{blame} p$ denotes a run-time type error caused by the failure of a coercion (projection) with blame label p .

Values, ranged over by V, W , include variables x , uncoerced values U , and coerced values $U\langle\langle d \rangle\rangle$. Uncoerced values, ranged over by U , include constants a of base types and lambda abstractions $\lambda x. M$. Unlike λC , where values can involve nested coercion applications, there is at most one coercion in a value – nested coercions will be composed. Coerced values $U\langle\langle d \rangle\rangle$ have two forms: injected values $U\langle\langle g; G! \rangle\rangle$ and wrapped functions $U\langle\langle s \rightarrow t \rangle\rangle$. The check of function coercion is delayed until wrapped functions are applied to a value [18, 13, 36]. We include variables as values for technical convenience in defining translations; for operational semantics, though, it is not necessary to do so because we consider evaluation of closed terms.

Unlike many other studies on coercion and blame calculi, we syntactically distinguish coerced values $U\langle\langle d \rangle\rangle$ from $U\langle d \rangle$ (similarly to Wadler and Findler [43]). This distinction plays an important role in our correctness proof; roughly speaking, without the distinction, $U\langle d \rangle\langle t \rangle$ would allow two different interpretations: an application of t to a value $U\langle d \rangle$ or two applications of d and t to a value U , which would result in different translation results. We also note that variables x are considered values, rather than uncoerced values, since they can be bound to coerced values at function calls. In other words, we ensure that values are closed under value substitution.

As usual, applications are left-associative and λ extends as far to the right as possible. We do not commit to a particular choice of precedence between function applications and coercion applications; we will always use parentheses to disambiguate terms like $M N\langle t \rangle$. The term $\lambda x. M$ binds x in M as usual. The definitions of free variables and α -equivalence of terms are standard, and thus we omit them. We identify α -equivalent terms.

The metavariable Γ ranges over *type environments*. A type environment is a sequence of pairs of a variable and its type.

2.2 Type System

We give the type system of λS , which consists of three judgments for *type consistency* $A \sim B$, *well-formed coercions* $c : A \rightsquigarrow B$, and *typing* $\Gamma \vdash_S M : A$. We use c to denote any kind of coercions. The inference rules (except for $A \sim B$) are shown in Figure 3. (We omit the subscript S on \vdash in rules, as some of them are reused for λS_1 .)

The type consistency relation $A \sim B$ is the least reflexive and symmetric and compatible relation that contains $A \sim \star$. As this is standard [36], we omit inference rules here. (We have them in the full version.)

The relation $c : A \rightsquigarrow B$ means that coercion c , which ranges over all kinds of coercions, converts a value from type A to type B . We often call A and B the source and target types of c , respectively. The rule (CT-ID) is for identity coercion id_A . The rule (CT-INJ) is for injection $G!$, which converts type G to type \star . The rule (CT-PROJ) is for projection $G?^p$, which converts type \star to type G . The rule (CT-FUN) is for function coercion $c_1 \rightarrow c_2$. If its argument coercion c_1 converts type A' to type A and its return-value coercion c_2 converts type B to type B' , then function coercion $c_1 \rightarrow c_2$ converts type $A \rightarrow B$ to type $A' \rightarrow B'$. In other words, function coercions are contravariant in their argument coercions and covariant in return-value coercions. The rule (CT-FAIL) is for failure coercion \perp^{GpH} . Here, the source

$$\begin{array}{c}
 \text{Well-formed coercions} \quad \boxed{c : A \rightsquigarrow B} \\
 \\
 \frac{}{G! : G \rightsquigarrow \star} \text{CT-INJ} \quad \frac{}{G?^p : \star \rightsquigarrow G} \text{CT-PROJ} \quad \frac{c_1 : A' \rightsquigarrow A \quad c_2 : B \rightsquigarrow B'}{c_1 \rightarrow c_2 : A \rightarrow B \rightsquigarrow A' \rightarrow B'} \text{CT-FUN} \\
 \\
 \frac{}{\text{id}_A : A \rightsquigarrow A} \text{CT-ID} \quad \frac{c_1 : A \rightsquigarrow B \quad c_2 : B \rightsquigarrow C}{(c_1; c_2) : A \rightsquigarrow C} \text{CT-SEQ} \quad \frac{A \neq \star \quad A \sim G \quad G \neq H}{\perp^{GpH} : A \rightsquigarrow B} \text{CT-FAIL} \\
 \\
 \text{Term typing} \quad \boxed{\Gamma \vdash_S M : A} \\
 \\
 \frac{}{\Gamma \vdash a : \text{ty}(a)} \text{T-CONST} \quad \frac{\text{ty}(op) = \iota_1 \rightarrow \iota_2 \rightarrow \iota \quad \Gamma \vdash M : \iota_1 \quad \Gamma \vdash N : \iota_2}{\Gamma \vdash op(M, N) : \iota} \text{T-OP} \\
 \\
 \frac{(\lambda x. A) \in \Gamma}{\Gamma \vdash x : A} \text{T-VAR} \quad \frac{\Gamma, x : A \vdash M : B}{\Gamma \vdash \lambda x. M : A \rightarrow B} \text{T-ABS} \quad \frac{\Gamma \vdash M : A \rightarrow B \quad \Gamma \vdash N : A}{\Gamma \vdash MN : B} \text{T-APP} \\
 \\
 \frac{\Gamma \vdash M : A \quad s : A \rightsquigarrow B}{\Gamma \vdash M\langle s \rangle : B} \text{T-CRC} \quad \frac{\emptyset \vdash U : A \quad d : A \rightsquigarrow B}{\emptyset \vdash U\langle\langle d \rangle\rangle : B} \text{T-CRCV} \quad \frac{}{\emptyset \vdash \text{blame } p : A} \text{T-BLAME}
 \end{array}$$

■ **Figure 3** Typing rules of $\lambda\mathcal{S}$.

type is not necessarily G but can be any nondynamic type A consistent with G because the source type of a failure coercion may change during coercion composition. For example, the following judgments are derivable:

$$\begin{array}{l}
 (\text{id}_{\text{int}}; \text{int}!) \rightarrow (\text{int}?^p; \text{id}_{\text{int}}) : \star \rightarrow \star \quad \rightsquigarrow \text{int} \rightarrow \text{int} \\
 \perp^{\star \rightarrow \star p \text{int}} : \text{int} \rightarrow \text{bool} \quad \rightsquigarrow \text{int}
 \end{array}$$

Proposition 1 below, which is about the source and target types of intermediate coercions and ground coercions, is useful to understand the syntactic structure of space-efficient coercions. In particular, it states that neither the source nor target type of ground coercions g is the type \star .

► **Proposition 1** (Source and Target Types).

1. If $i : A \rightsquigarrow B$ then $A \neq \star$.
2. If $g : A \rightsquigarrow B$, then $A \neq \star$ and $B \neq \star$ and $A \sim G$ and $G \sim B$ for some unique G .

The judgment $\Gamma \vdash_S M : A$ means that the $\lambda\mathcal{S}$ -term M is given type A under type environment Γ . When clear from the context, we sometimes write \vdash for \vdash_S with the subscript S omitted. We adopt similar conventions for other relations (such as \mapsto_S) introduced later.

The rules (T-CONST), (T-OP), (T-VAR), (T-ABS), and (T-APP) are standard. Here, $\text{ty}(a)$ maps constant a to a base type ι , and $\text{ty}(op)$ maps binary operator op to a (first-order) function type $\iota_1 \rightarrow \iota_2 \rightarrow \iota$. The rule (T-CRC) states that if M is given type A and space-efficient coercion s converts type A to B , then coercion application $M\langle s \rangle$ is given type B . The rule (T-CRCV) is similar to (T-CRC), but for coerced values $U\langle\langle d \rangle\rangle$. The rule (T-BLAME) allows $\text{blame } p$ to have an arbitrary type A . Here, type environments are always empty \emptyset in (T-CRCV) and (T-BLAME). It is valid because the terms $U\langle\langle d \rangle\rangle$ and $\text{blame } p$ arise only during evaluation, which runs a closed term. In other words, these terms are not written by programmers in the surface language, and also they do not appear as the result of coercion insertion.

Coercion composition

$$\boxed{s \ ; \ t = s'}$$

$\text{id}_\star \ ; \ t = t$	CC-IDDYNL	$(G^{?^p}; i) \ ; \ t = G^{?^p}; (i \ ; \ t)$	CC-PROJL
$(g; G!) \ ; \ \text{id}_\star = g; G!$	CC-INJID	$(g; G!) \ ; \ (G^{?^p}; i) = g \ ; \ i$	CC-COLLAPSE
$\perp^{G^pH} \ ; \ s = \perp^{G^pH}$	CC-FAILL	$(g; G!) \ ; \ (H^{?^p}; i) = \perp^{G^pH}$ (if $G \neq H$)	CC-CONFLICT
$g \ ; \ \perp^{G^pH} = \perp^{G^pH}$	CC-FAILR	$g \ ; \ (h; H!) = (g \ ; \ h); H!$	CC-INJR
$\text{id}_A \ ; \ g = g$ (if $A \neq \star$)	CC-IDL	$g \ ; \ \text{id}_A = g$ (if $A \neq \star, g \neq \text{id}_A$)	CC-IDR
$(s \rightarrow t) \ ; \ (s' \rightarrow t') =$		$\begin{cases} \text{id}_{A \rightarrow B} & \text{if } s' \ ; \ s = \text{id}_A \text{ and } t \ ; \ t' = \text{id}_B \\ (s' \ ; \ s) \rightarrow (t \ ; \ t') & \text{otherwise} \end{cases}$	CC-FUN

■ **Figure 4** Coercion composition rules of $\lambda\mathcal{S}$.

Evaluation contexts

$$\mathcal{E} ::= \mathcal{F} \mid \mathcal{F}[\square \langle s \rangle] \quad \mathcal{F} ::= \square \mid \mathcal{E}[\text{op}(\square, M)] \mid \mathcal{E}[\text{op}(V, \square)] \mid \mathcal{E}[\square M] \mid \mathcal{E}[V \square]$$

Reduction

$$\boxed{M \xrightarrow{e} N} \quad \boxed{M \xrightarrow{c} N}$$

$\text{op}(a, b) \xrightarrow{e} \delta(\text{op}, a, b)$	R-OP	$U \langle \text{id}_A \rangle \xrightarrow{c} U$	R-ID
$(\lambda x. M) V \xrightarrow{e} M[x := V]$	R-BETA	$U \langle \perp^{G^pH} \rangle \xrightarrow{c} \text{blame } p$	R-FAIL
$(U \langle \langle s \rightarrow t \rangle \rangle) V \xrightarrow{e} (U \langle V \langle s \rangle \rangle) \langle t \rangle$	R-WRAP	$U \langle d \rangle \xrightarrow{c} U \langle \langle d \rangle \rangle$	R-CRC
		$M \langle s \rangle \langle t \rangle \xrightarrow{c} M \langle s \ ; \ t \rangle$	R-MERGE C
		$U \langle \langle d \rangle \rangle \langle t \rangle \xrightarrow{c} U \langle d \ ; \ t \rangle$	R-MERGE V

Evaluation

$$\boxed{M \mapsto_{S_1} N} \quad \boxed{M \xrightarrow{c}_{S_1} N}$$

$$\frac{M \xrightarrow{e} N}{\mathcal{E}[M] \mapsto_{S_1} \mathcal{E}[N]} \text{E-CTXE} \quad \frac{M \xrightarrow{c} N}{\mathcal{F}[M] \mapsto_{S_1} \mathcal{F}[N]} \text{E-CTXC} \quad \frac{\mathcal{E} \neq \square}{\mathcal{E}[\text{blame } p] \mapsto_{S_1} \text{blame } p} \text{E-ABORT}$$

■ **Figure 5** Reduction/evaluation rules of $\lambda\mathcal{S}$.

2.3 Operational Semantics

2.3.1 Coercion Composition

The coercion composition $s \ ; \ t$ is a recursive function that takes two space-efficient coercions and computes another space-efficient coercion corresponding to their sequential composition. We show the coercion composition rules in Figure 4. The function is defined in such a way that the form of the first coercion determines which rule to apply.

The rules (CC-IDDYNL) and (CC-PROJL) are applied if the first coercion is not an intermediate coercion. The rules (CC-INJID), (CC-COLLAPSE), (CC-CONFLICT), and (CC-FAILL) are applied if the first one is a (nonground) intermediate coercion, in which case another intermediate coercion is yielded. The rules (CC-COLLAPSE) and (CC-CONFLICT) deal with cases where an injection and a projection meet and perform tag checks. If type tags do not match, a failure coercion arises.

Failure coercions are necessary for eager coercion composition to preserve the behavior of $\lambda\mathcal{C}$. The term $M \langle G! \rangle \langle H^{?^p} \rangle$ (if $G \neq H$) in $\lambda\mathcal{C}$ evaluates to $\text{blame } p$ – *only after M evaluates to a value*. By contrast, the two coercions $G!$ and $H^{?^p}$ in the term $M \langle \text{id}_G; G! \rangle \langle H^{?^p}; \text{id}_H \rangle$

are eagerly composed in $\lambda\mathcal{S}$. Raising blame p immediately would not match the semantics of $\lambda\mathcal{C}$ because M may evaluate to another blame or even diverge, in which case p is not blamed. Thus, \perp^{GpH} must raise blame p only after M evaluates to a value.

The rules (CC-FAILR) and (CC-INJR) are applied if a ground coercion and an intermediate coercion are composed to another intermediate coercion. The rules (CC-FAILL) and (CC-FAILR) represent the propagation of a failure to the context, somewhat similarly to exceptions. The rule (CC-INJR) represents associativity of sequential compositions but \circledast is propagated to the inside.

The rules (CC-IDL), (CC-IDR), and (CC-FUN) are applied if two ground coercions are composed to another ground coercion. They are straightforward except that $\text{id}_A \rightarrow \text{id}_B$ has to be normalized to $\text{id}_{A \rightarrow B}$ (CC-FUN).

We present a few examples of coercion composition below:

$$\begin{aligned} (\text{id}_{\text{bool}}; \text{bool}!) \circledast (\text{bool}^?^p; \text{id}_{\text{bool}}) &= \text{id}_{\text{bool}} \circledast \text{id}_{\text{bool}} = \text{id}_{\text{bool}} \\ (\text{id}_{\star \rightarrow \star}; (\star \rightarrow \star)!) \circledast (\text{int}^?^p; \text{id}_{\text{int}}) &= \perp^{\star \rightarrow \star p \text{int}} \\ ((\iota^?^p; \text{id}_\iota) \rightarrow (\text{id}_{\iota'}; \iota'!)) \circledast ((\text{id}_\iota; \iota!) \rightarrow \text{id}_\star) &= ((\text{id}_\iota; \iota!) \circledast (\iota^?^p; \text{id}_\iota)) \rightarrow ((\text{id}_{\iota'}; \iota'!) \circledast \text{id}_\star) \\ &= \text{id}_\iota \rightarrow (\text{id}_{\iota'}; \iota'!) \end{aligned}$$

These examples involve situations where an injection meets a projection by (CC-COLLAPSE) or (CC-CONFLICT). The third example is by (CC-FUN).

$$\begin{aligned} (\iota^?^p; \text{id}_\iota) \circledast (\text{id}_\iota; \iota!) &= \iota^?^p; (\text{id}_\iota \circledast (\text{id}_\iota; \iota!)) = \iota^?^p; ((\text{id}_\iota \circledast \text{id}_\iota); \iota!) = \iota^?^p; (\text{id}_\iota; \iota!) \\ (\text{id}_\iota; \iota!) \circledast (\iota^?^p; (\text{id}_\iota; \iota!)) &= \text{id}_\iota \circledast (\text{id}_\iota; \iota!) = (\text{id}_\iota \circledast \text{id}_\iota); \iota! = \text{id}_\iota; \iota! \end{aligned}$$

As the fourth example shows, a projection followed by an injection does not collapse since the projection might fail. Such a coercion is simplified when it is preceded by another injection (the fifth example).

The following lemma states that composition is defined for two well-formed coercions with matching target and source types.

► **Lemma 2.** *If $s : A \rightsquigarrow B$ and $t : B \rightsquigarrow C$, then $(s \circledast t) : A \rightsquigarrow C$.*

2.3.2 Evaluation

We give a small-step operational semantics to $\lambda\mathcal{S}$ consisting of two relations on closed terms: the reduction relation $M \longrightarrow_{\mathcal{S}} N$ for basic computation, and the evaluation relation $M \mapsto_{\mathcal{S}} N$ for computing subterms and raising errors.

We show the reduction rules and the evaluation rules of $\lambda\mathcal{S}$ in Figure 5. The reduction/evaluation rules are labeled either e or c. The label e is for essential computation, and the label c is for coercion applications. As we see later, this distinction is important in our correctness proof. We write $\longrightarrow_{\mathcal{S}}$ for $\xrightarrow{e}_{\mathcal{S}} \cup \xrightarrow{c}_{\mathcal{S}}$, and $\mapsto_{\mathcal{S}}$ for $\xrightarrow{e}_{\mathcal{S}} \cup \xrightarrow{c}_{\mathcal{S}}$. We sometimes call $\xrightarrow{e}_{\mathcal{S}}$ and $\xrightarrow{c}_{\mathcal{S}}$ e-evaluation and c-evaluation, respectively.

The rule (R-OP) applies to primitive operations. Here, δ is a (partial) function that takes an operator op and two constants a_1, a_2 , and returns the resulting constant of the primitive operation. We assume that if $ty(op) = \iota_1 \rightarrow \iota_2 \rightarrow \iota$ and $ty(a_1) = \iota_1$ and $ty(a_2) = \iota_2$, then $\delta(op, a_1, a_2) = a$ and $ty(a) = \iota$ for some constant a .

The rule (R-BETA) performs the standard call-by-value β -reduction. We write $M[x := V]$ for capture-avoiding substitution of V for free occurrences of x in M . The definition of substitution is standard and thus omitted.

The rule (R-WRAP) applies to applications of wrapped function $U\langle\langle s \rightarrow t \rangle\rangle$ to value V . In this case, we first apply coercion s on the argument to V , and get $V\langle s \rangle$. We next apply function U to $V\langle s \rangle$, and get $U(V\langle s \rangle)$. We then apply coercion t on the returned value, hence $(U(V\langle s \rangle))\langle t \rangle$.

The rule (R-ID) represents that identity coercion id_A returns the input value U as it is. The rule (R-FAIL) applies to applications of failure coercion \perp^{GpH} to uncoerced value U , which reduces to $\text{blame } p$. The rule (R-CRC) applies to applications $U\langle d \rangle$ of delayed coercion d to uncoerced value U , which reduces to a coerced value $U\langle\langle d \rangle\rangle$.

The rules (R-MERGE_C) and (R-MERGE_V) apply to two consecutive coercion applications, and the two coercions are merged by the composition operation. These rules are key to space efficiency. Thanks to (R-MERGE_V), we can assume that there is at most one coercion in a value. Since $d \circ t$ may or may not be a delayed coercion, the right-hand side has to be $U\langle d \circ t \rangle$, rather than $U\langle\langle d \circ t \rangle\rangle$. The outermost nested coercion applications are merged by (R-MERGE_C).

Now, we explain *evaluation contexts*, ranged over by \mathcal{E} , shown in the top of Figure 5. Following Siek et al. [37], we define them in the so-called “inside-out” style [11, 9]. Evaluation contexts represent that function calls in $\lambda\mathcal{S}$ are call-by-value and that primitive operations and function applications are evaluated from left to right. The grammar is mutually recursive with \mathcal{F} , which stands for evaluation contexts whose innermost frames are not a coercion application, whereas \mathcal{E} may contain a coercion application as the innermost frame.⁴ Careful inspection will reveal that both \mathcal{E} and \mathcal{F} contain no consecutive coercion applications. As usual, we write $\mathcal{E}[M]$ for the term obtained by replacing the hole in \mathcal{E} with M , similarly for $\mathcal{F}[M]$. (We omit their definitions.)

We present a few examples of evaluation contexts below:

$$\begin{aligned} \mathcal{F}_1 &= \square & \mathcal{E}_1 &= \mathcal{F}_1[\square\langle s \rangle] = \square\langle s \rangle \\ \mathcal{F}_2 &= \mathcal{E}_1[V\square] = (V\square)\langle s \rangle & \mathcal{E}_2 &= \mathcal{F}_2[\square\langle t \rangle] = (V(\square\langle t \rangle))\langle s \rangle \\ \mathcal{F}_3 &= \mathcal{E}_2[\square M] = (V((\square M)\langle t \rangle))\langle s \rangle \end{aligned}$$

We then come back to evaluation rules: The rules (E-CTX_E) and (E-CTX_C) enable us to evaluate the subterm in an evaluation context. Here, (E-CTX_C) requires that computation of coercion applications is only performed under contexts \mathcal{F} – otherwise, the innermost frame may be a coercion application, in which case (R-MERGE_C) has to be applied first. For example, $U\langle d \rangle\langle t \rangle$ reduces to $U\langle d \circ t \rangle$ rather than $U\langle\langle d \rangle\rangle\langle t \rangle$. The rule (E-ABORT) halts the evaluation of a program if it raises blame.

► **Example 3.** Let U be $\lambda x. (x\langle \text{int}^?^p \rangle + 2)\langle \text{int}! \rangle$. Term $((U\langle \text{int}! \rightarrow \text{int}^?^p \rangle) 3)\langle \text{int}! \rangle$ evaluates to $5\langle\langle \text{int}! \rangle\rangle$ as follows:

$$\begin{aligned} & ((U\langle \text{int}! \rightarrow \text{int}^?^p \rangle) 3)\langle \text{int}! \rangle \\ & \mapsto^* (U(3\langle \text{int}! \rangle))\langle \text{int}^?^p \rangle\langle \text{int}! \rangle && \text{by (R-CRC), (R-WRAP)} \\ & \mapsto (U(3\langle \text{int}! \rangle))\langle \text{int}^?^p; \text{id}; \text{int}! \rangle && \text{by (R-MERGE}_C) \\ & \mapsto^* (3\langle\langle \text{int}! \rangle\rangle)\langle \text{int}^?^p \rangle + 2)\langle \text{int}! \rangle\langle \text{int}^?^p; \text{id}; \text{int}! \rangle && \text{by (R-CRC), (R-BETA)} \\ & \mapsto^* (3\langle \text{id} \rangle + 2)\langle \text{int}! \rangle && \text{by (R-MERGE}_C), \text{(R-MERGE}_V) \\ & \mapsto^* 5\langle\langle \text{int}! \rangle\rangle && \text{by (R-ID), (R-OP), (R-CRC)}. \end{aligned}$$

⁴ $\mathcal{F}[\square\langle s \rangle]$ (instead of $\mathcal{F}[\square\langle f \rangle]$) in the definition of \mathcal{E} fixes a problem in Siek et al. [37] that an identity coercion applied to a nonvalue gets stuck (personal communication).

2.4 Properties

We state a few important properties of $\lambda\mathcal{S}$, including determinacy of the evaluation relation and type safety via progress and preservation [46]. We write $\mapsto_{\mathcal{S}}^*$ for the reflexive and transitive closure of $\mapsto_{\mathcal{S}}$, and $\mapsto_{\mathcal{S}}^+$ for the transitive closure of $\mapsto_{\mathcal{S}}$. We say that $\lambda\mathcal{S}$ -term M *diverges*, denoted by $M \uparrow_{\mathcal{S}}$, if there exists an infinite evaluation sequence from M .

Proofs of the stated properties are in the full version.

- ▶ **Lemma 4** (Determinacy). *If $M \mapsto_{\mathcal{S}} N$ and $M \mapsto_{\mathcal{S}} N'$, then $N = N'$.*
- ▶ **Theorem 5** (Progress). *If $\emptyset \vdash_{\mathcal{S}} M : A$, then one of the following holds: (1) $M \mapsto_{\mathcal{S}} M'$ for some M' ; (2) $M = V$ for some V ; or (3) $M = \text{blame } p$ for some p .*
- ▶ **Theorem 6** (Preservation). *If $\emptyset \vdash_{\mathcal{S}} M : A$ and $M \mapsto_{\mathcal{S}} N$, then $\emptyset \vdash_{\mathcal{S}} N : A$.*
- ▶ **Corollary 7** (Type Safety). *If $\emptyset \vdash_{\mathcal{S}} M : A$, then one of the following holds: (1) $M \mapsto_{\mathcal{S}}^* V$ and $\emptyset \vdash_{\mathcal{S}} V : A$ for some V ; (2) $M \mapsto_{\mathcal{S}}^* \text{blame } p$ for some p ; or (3) $M \uparrow_{\mathcal{S}}$.*

3 Space-Efficient First-Class Coercion Calculus

In this section, we introduce $\lambda\mathcal{S}_1$, a new space-efficient coercion calculus with first-class coercions; $\lambda\mathcal{S}_1$ serves as the target calculus of the translation into coercion-passing style. The design of $\lambda\mathcal{S}_1$ is tailored to coercion-passing style and, as a result, first-class coercions are not as general as one might expect: for example, coercions for coercions are restricted to identity coercions (e.g., $\text{id}_{l \rightsquigarrow l}$).

Since coercions are first-class in $\lambda\mathcal{S}_1$, the use of (space-efficient) coercions s is not limited to coercion applications $M\langle s \rangle$; they can be passed to a function as an argument, for example. We equip $\lambda\mathcal{S}$ with the infix (object-level) operator $M ;; N$ to compute the composition of two coercions: if M and N evaluate to coercions s and t , respectively, then $M ;; N$ reduces to their composition $s \S t$, which is another space-efficient coercion. The type of (first-class) coercions from A to B is written $A \rightsquigarrow B$.⁵

In $\lambda\mathcal{S}_1$, every function abstraction takes two arguments, one of which is a parameter for a continuation coercion to be applied to the value returned from this abstraction. For example, $\lambda x. 1$ in $\lambda\mathcal{S}$ corresponds to $\lambda(x, \kappa). 1\langle \kappa \rangle$ in $\lambda\mathcal{S}_1$ — here, κ is a coercion parameter. Correspondingly, a function application takes the form $M(N, L)$, which calls function M with an argument pair (N, L) , in which L is a coercion argument, which is applied to the value returned from M . For example, $(f\ 3)\langle s \rangle$ in $\lambda\mathcal{S}$ corresponds to $f(3, s)$ in $\lambda\mathcal{S}_1$; $(f\ 3)$ (without a coercion application) corresponds to $f(3, \text{id})$.

The type of a function abstraction in $\lambda\mathcal{S}_1$ is written $A \Rightarrow B$, which means that the type of the first argument is the type A and the source type of the second coercion argument is B . An abstraction is polymorphic over the target type of the coercion argument; so, if a function of type $A \Rightarrow B$ is applied to a pair of A and $B \rightsquigarrow C$, then the type of the application will be C . Polymorphism is useful – and in fact required – for coercion-passing translation to work because coercions with different target types may be passed to calls to the same function in $\lambda\mathcal{S}$. Intuitively, $A \Rightarrow B$ means $\forall X. (A \times (B \rightsquigarrow X)) \rightarrow X$ but we do not introduce \forall -types explicitly because our use of \forall is limited to the target-type polymorphism. However, we do have to introduce type variables for typing function abstractions.

⁵ In $\lambda\mathcal{S}$, \rightsquigarrow is the symbol used in the three-place judgment form $c : A \rightsquigarrow B$, whereas \rightsquigarrow is also a type constructor in $\lambda\mathcal{S}_1$.

Variables	x, y, κ	Type variables	X, Y
Types	$A, B, C ::= \star \mid \iota \mid A \rightsquigarrow B \mid A \Rightarrow B \mid X$		
Ground types	$G, H ::= \iota \mid \star \Rightarrow \star$		
Space-efficient coercions	$s, t ::= \text{id}_\star \mid G^{?^p}; i \mid i$		
Intermediate coercions	$i ::= g; G! \mid g \mid \perp^{GpH}$		
Ground coercions	$g, h ::= \text{id}_A \text{ (if } A \neq \star) \mid s \Rightarrow t \text{ (if } s \neq \text{id or } t \neq \text{id)}$		
Delayed coercions	$d ::= g; G! \mid s \Rightarrow t \text{ (if } s \neq \text{id or } t \neq \text{id)}$		
Terms	$L, M, N ::= V \mid \text{op}(M, N) \mid L(M, N) \mid \text{let } x = M \text{ in } N$ $\mid M ;; N \mid M \langle N \rangle \mid \text{blame } p$		
Values	$V, W, K ::= x \mid U \mid U \langle d \rangle$		
Uncoerced values	$U ::= a \mid \lambda(x, \kappa). M \mid s$		
Type environments	$\Gamma ::= \emptyset \mid \Gamma, x : A$		

■ **Figure 6** Syntax of $\lambda\mathcal{S}_1$.

Following the change to function types, function coercions in $\lambda\mathcal{S}_1$ take the form $s \Rightarrow t$. Roughly speaking, its meaning is the same: it coerces an input to a function by s and coerces an output by t . However, due to the coercion passing semantics, there is slight change in how t is used at a function call. Consider $f \langle \langle s \Rightarrow t \rangle \rangle$, i.e., coercion-passing function f wrapped by coercion $s \Rightarrow t$. If the wrapped function is applied to (V, t') , V is coerced by s before passing to f as in $\lambda\mathcal{S}$; instead of coercing the return value by t , however, t is prepended to t' and passed to f (together with the coerced V) so that the return value is coerced by t and then t' . In the reduction rule, prepending t to t' is represented by composition $t ;; t'$.

3.1 Syntax

We show the syntax of $\lambda\mathcal{S}_1$ in Figure 6. We reuse the same metavariables from $\lambda\mathcal{S}$. We also use κ for variables, and K for values.

We replace $A \rightarrow B$ with $A \Rightarrow B$ and add $A \rightsquigarrow B$ and type variables to types. The syntax for ground types and space-efficient, intermediate, ground, and delayed coercions is the same except that \rightarrow is replaced with \Rightarrow , similarly to types. As we have mentioned, we replace abstractions and applications with two-argument versions. We also add let-expressions (although they could be introduced as derived forms) and coercion composition $M ;; N$. The syntax for coercion applications is now $M \langle N \rangle$, where N is a general term (of type $A \rightsquigarrow B$). Uncoerced values now include space-efficient coercions.

The term $\lambda(x, \kappa). M$ binds x and κ in M , and the term $\text{let } x = M \text{ in } N$ binds x in N . The definitions of free variables and α -equivalence of terms are standard, and thus we omit them. We identify α -equivalent terms.

The definition of type environments, ranged over by Γ , is the same as $\lambda\mathcal{S}$.

3.2 Type System

Figure 7 shows the main typing rules of $\lambda\mathcal{S}_1$, which are a straightforward adaption from $\lambda\mathcal{S}$.

The relation $c : A \rightsquigarrow B$ is mostly the same as that of $\lambda\mathcal{S}$. We replace the rule (CT-FUN) as shown. As in $\lambda\mathcal{S}$, function coercions are contravariant in their argument coercions and covariant in their return-value coercions.

Well-formed coercions (replacement)

$$\boxed{c : A \rightsquigarrow B}$$

$$\frac{c_1 : A' \rightsquigarrow A \quad c_2 : B \rightsquigarrow B'}{c_1 \Rightarrow c_2 : A \Rightarrow B \rightsquigarrow A' \Rightarrow B'} \text{CT-FUN}$$

Term typing (excerpt)

$$\boxed{\Gamma \vdash_{\mathcal{S}_1} M : A}$$

$$\frac{s : A \rightsquigarrow B}{\Gamma \vdash s : A \rightsquigarrow B} \text{T-CRCN} \quad \frac{\Gamma \vdash M : A \rightsquigarrow B \quad \Gamma \vdash N : B \rightsquigarrow C}{\Gamma \vdash M ;; N : A \rightsquigarrow C} \text{T-CMP}$$

$$\frac{\Gamma \vdash M : A \quad \Gamma \vdash N : A \rightsquigarrow B}{\Gamma \vdash M \langle N \rangle : B} \text{T-CRC} \quad \frac{\emptyset \vdash U : A \quad \emptyset \vdash d : A \rightsquigarrow B}{\emptyset \vdash U \langle\langle d \rangle\rangle : B} \text{T-CRCV}$$

$$\frac{\Gamma, x : A, \kappa : B \rightsquigarrow X \vdash M : X \quad (X \text{ does not appear in } \Gamma, A, B)}{\Gamma \vdash \lambda(x, \kappa). M : A \Rightarrow B} \text{T-ABS}$$

$$\frac{\Gamma \vdash M : A \quad \Gamma, x : A \vdash N : B}{\Gamma \vdash \text{let } x = M \text{ in } N : B} \text{T-LET} \quad \frac{\Gamma \vdash L : A \Rightarrow B \quad \Gamma \vdash M : A \quad \Gamma \vdash N : B \rightsquigarrow C}{\Gamma \vdash L(M, N) : C} \text{T-APP}$$

■ **Figure 7** Typing rules of $\lambda\mathcal{S}_1$.

The judgment $\Gamma \vdash_{\mathcal{S}_1} M : A$ means that term M of $\lambda\mathcal{S}_1$ has type A under type environment Γ . The rules (T-CONST), (T-OP), (T-VAR), and (T-BLAME) are the same as $\lambda\mathcal{S}$, and so we omit them. The rule (T-LET) is standard.

The rules (T-ABS) and (T-APP) look involved but the intuition that $A \Rightarrow B$ corresponds to $\forall X. (A \times (B \rightsquigarrow X)) \rightarrow X$ should help to understand them. The rule (T-ABS) assigns type $A \Rightarrow B$ to an abstraction $\lambda(x, \kappa). M$ if the body is well typed under the assumption that x is of type A and κ is of type $B \rightsquigarrow X$ for fresh X . The type variable X must not appear in Γ, A, B so that the target type can be polymorphic at call sites. The rule (T-APP) for applications is already explained.

The rule (T-CRCN) assigns type $A \rightsquigarrow B$ to space-efficient coercion s if it converts a value from type A to type B . The rules (T-CRC) and (T-CRCV) are similar to the corresponding rules of $\lambda\mathcal{S}$, but adjusted to first-class coercions.

3.3 Operational Semantics

The composition function $s \circledast t$ is mostly the same as that of $\lambda\mathcal{S}$. We only replace (CC-FUN) as shown in Figure 8.

Similarly to $\lambda\mathcal{S}$, we give a small-step operational semantics to $\lambda\mathcal{S}_1$ consisting of two relations on closed terms: the reduction relation $M \longrightarrow_{\mathcal{S}_1} N$ and the evaluation relation $M \mapsto_{\mathcal{S}_1} N$. We show the reduction/evaluation rules of $\lambda\mathcal{S}_1$ in Figure 8. As in $\lambda\mathcal{S}$, they are labeled either e or c. We write $\longrightarrow_{\mathcal{S}_1}$ for $\xrightarrow{e}_{\mathcal{S}_1} \cup \xrightarrow{c}_{\mathcal{S}_1}$, and $\mapsto_{\mathcal{S}_1}$ for $\xrightarrow{e}_{\mathcal{S}_1} \cup \xrightarrow{c}_{\mathcal{S}_1}$.

The rules (R-OP) and (R-BETA) are standard. Note that (R-BETA) is adjusted for pair arguments. We write $M[x := V, \kappa := K]$ for capture-avoiding simultaneous substitution of V and K for x and κ , respectively, in M .

The rule (R-WRAP) applies to applications of wrapped function $U \langle\langle s \Rightarrow t \rangle\rangle$ to value V . Since coercion s is for function arguments, it is applied to V , as in $\lambda\mathcal{S}$. Additionally, we compose coercion t on the return value with continuation coercion W . Thus, $V \langle s \rangle$ and $t ;; W$ are passed to function U . Note that we use a let expression to evaluate the second argument $t ;; W$ before $V \langle s \rangle$. It is a necessary adjustment for the semantics of $\lambda\mathcal{S}$ and $\lambda\mathcal{S}_1$ to match.

Coercion composition (replacement)

$$s \ddot{;} t = s'$$

$$(s \Rightarrow t) \ddot{;} (s' \Rightarrow t') = \begin{cases} \text{id}_{A \Rightarrow B} & \text{if } s' \ddot{;} s = \text{id}_A \text{ and } t \ddot{;} t' = \text{id}_B \\ (s' \ddot{;} s) \Rightarrow (t \ddot{;} t') & \text{otherwise} \end{cases} \quad \text{CC-FUN}$$

Evaluation contexts

$$\mathcal{E} ::= \square \mid \mathcal{E}[\square(M, N)] \mid \mathcal{E}[V(\square, N)] \mid \mathcal{E}[V(W, \square)] \mid \mathcal{E}[op(\square, M)] \mid \mathcal{E}[op(V, \square)] \\ \mid \mathcal{E}[\text{let } x = \square \text{ in } M] \mid \mathcal{E}[\square ;; M] \mid \mathcal{E}[V ;; \square] \mid \mathcal{E}[\square \langle M \rangle] \mid \mathcal{E}[V \langle \square \rangle]$$

Reduction

$$M \xrightarrow{e}_{S_1} N \quad M \xrightarrow{c}_{S_1} N$$

$$\begin{array}{ll} op(a, b) \xrightarrow{e} \delta(op, a, b) & \text{R-OP} \\ (\lambda(x, \kappa). M)(V, W) \xrightarrow{e} M[x := V, \kappa := W] & \text{R-BETA} \\ (U \langle \langle s \Rightarrow t \rangle \rangle)(V, W) \xrightarrow{e} \text{let } \kappa = t ;; W \text{ in } U(V \langle s \rangle, \kappa) & \text{R-WRAP} \end{array}$$

$$\begin{array}{ll} \text{let } x = V \text{ in } M \xrightarrow{c} M[x := V] & \text{R-LET} & s ;; t \xrightarrow{c} s \ddot{;} t & \text{R-CMP} \\ U \langle \text{id}_A \rangle \xrightarrow{c} U & \text{R-ID} & U \langle \perp^{GpH} \rangle \xrightarrow{c} \text{blame } p & \text{R-FAIL} \\ U \langle d \rangle \xrightarrow{c} U \langle \langle d \rangle \rangle & \text{R-CRC} & U \langle \langle d \rangle \rangle \langle t \rangle \xrightarrow{c} U \langle d ;; t \rangle & \text{R-MERGEV} \end{array}$$

Evaluation

$$M \vdash^e_{S_1} N \quad M \vdash^c_{S_1} N$$

$$\frac{M \xrightarrow{x} N \quad \mathcal{X} \in \{\mathbf{e}, \mathbf{c}\}}{\mathcal{E}[M] \vdash^x \mathcal{E}[N]} \text{E-CTX} \quad \frac{\mathcal{E} \neq \square}{\mathcal{E}[\text{blame } p] \vdash^e \text{blame } p} \text{E-ABORT}$$

■ **Figure 8** Reduction/evaluation rules of λS_1 .

The rule (R-LET) is standard; it is labeled as **c** because we use let-expressions only for coercion compositions. The rule (R-CMP) applies to coercion compositions $s ;; t$, which is evaluated by meta-level coercion composition function $s \ddot{;} t$. The rules (R-ID), (R-FAIL), (R-CRC), and (R-MERGEV) are the same as λS .

Evaluation contexts, ranged over by \mathcal{E} , are defined also in Figure 8. In contrast to λS , evaluation contexts are standard in λS_1 . The definition represents that function calls in λS_1 are call-by-value, and primitive operations, function applications, coercion compositions, and coercion applications are all evaluated from left to right.

We then come back to evaluation rules: The evaluation rules (E-CTX) and (E-ABORT) are the same as λS . (However, evaluation contexts in (E-CTX) are more straightforward in λS_1 .)

Finally, we should emphasize that we no longer need (R-MERGEV) in λS_1 . So, λS_1 is an ordinary call-by-value language and its semantics should be easy to implement.

► **Example 8.** Let U be $\lambda(x, \kappa). \text{let } \kappa' = \text{int}! ;; \kappa \text{ in } (x \langle \text{int}^{?p} \rangle + 2) \langle \kappa' \rangle$, which corresponds to the λS -term $\lambda x. (x \langle \text{int}^{?p} \rangle + 2) \langle \text{int}! \rangle$ in Example 3. In fact, we will obtain this term as a result of our coercion-passing translation defined in the next section. The term $(U \langle \text{int}! \Rightarrow \text{int}^{?p} \rangle)(3, \text{int}!)$ evaluates to $5 \langle \langle \text{int}! \rangle \rangle$ as follows:

$$\begin{aligned}
& (U\langle \text{int!} \Rightarrow \text{int}^?^p \rangle) (3, \text{int!}) \\
& \longmapsto^* \text{let } \kappa'' = \text{int}^?^p ;; \text{int! in } U (3\langle \text{int!} \rangle, \kappa'') && \text{by (R-CRC), (R-WRAP)} \\
& \longmapsto \text{let } \kappa'' = \text{int}^?^p; \text{id}; \text{int! in } U (3\langle \text{int!} \rangle, \kappa'') && \text{by (R-CMP)} \\
& \longmapsto^* U (3\langle\langle \text{int!} \rangle\rangle, (\text{int}^?^p; \text{id}; \text{int!})) && \text{by (R-LET), (R-CRC)} \\
& \longmapsto \text{let } \kappa' = \text{int!} ;; (\text{int}^?^p; \text{id}; \text{int!}) \text{ in } (3\langle\langle \text{int!} \rangle\rangle\langle \text{int}^?^p \rangle + 2)\langle \kappa' \rangle && \text{by (R-BETA)} \\
& \longmapsto^* (3\langle\langle \text{int!} \rangle\rangle\langle \text{int}^?^p \rangle + 2)\langle \text{int!} \rangle && \text{by (R-CMP), (R-LET)} \\
& \longmapsto^* 5\langle\langle \text{int!} \rangle\rangle && \text{by (R-MERGEV), (R-ID), (R-OP), (R-CRC)}
\end{aligned}$$

It is easy to see that the steps by (R-MERGEV) in Example 3 are simulated by (R-CMP) followed by (R-LET).

3.4 Properties

We state a few properties of $\lambda\mathcal{S}_1$ below. Their proofs are in the full version.

- **Lemma 9** (Determinacy). *If $M \mapsto_{\mathcal{S}_1} N$ and $M \mapsto_{\mathcal{S}_1} N'$, then $N = N'$.*
- **Theorem 10** (Progress). *If $\emptyset \vdash_{\mathcal{S}_1} M : A$, then one of the following holds: (1) $M \mapsto_{\mathcal{S}_1} M'$ for some M' ; (2) $M = V$ for some V ; or (3) $M = \text{blame } p$ for some p .*
- **Theorem 11** (Preservation). *If $\emptyset \vdash_{\mathcal{S}_1} M : A$ and $M \mapsto_{\mathcal{S}_1} N$, then $\emptyset \vdash_{\mathcal{S}_1} N : A$.*
- **Corollary 12** (Type Safety). *If $\emptyset \vdash_{\mathcal{S}_1} M : A$, then one of the following holds: (1) $M \mapsto_{\mathcal{S}_1}^* V$ and $\emptyset \vdash_{\mathcal{S}_1} V : A$ for some V ; (2) $M \mapsto_{\mathcal{S}_1}^* \text{blame } p$ for some p ; or (3) $M \uparrow_{\mathcal{S}_1}$.*

4 Translation into Coercion-Passing Style

In this section, we formalize a translation into coercion-passing style as a translation from $\lambda\mathcal{S}$ to $\lambda\mathcal{S}_1$ and state its correctness. As its name suggests, this translation is similar to transformations into continuation-passing style (CPS transformations) for the call-by-value λ -calculus [28].

4.1 Definition of Translation

We give the translation into coercion-passing style by the translation rules presented in Figure 9. In order to distinguish metavariables of $\lambda\mathcal{S}$ and $\lambda\mathcal{S}_1$, we often use [blue](#) for the source calculus $\lambda\mathcal{S}$. When we need static type information in translation rules, we write M^A to indicate that term M has type A . Thus, strictly speaking, the translation is defined for type derivations in $\lambda\mathcal{S}$.

Translations for types $\Psi(A)$ and coercions $\Psi(s)$ are very straightforward, thanks to the special type/coercion constructor \Rightarrow : they just recursively replace \rightarrow with \Rightarrow .

Value translation $\Psi(V)$ and term translation $\mathcal{K}[[M]]K$ are defined in a mutually recursive manner. In $\mathcal{K}[[M]]K$, M is a $\lambda\mathcal{S}$ -term whereas K is a $\lambda\mathcal{S}_1$ -term, which is either a variable or a $\lambda\mathcal{S}_1$ -coercion. $\mathcal{K}[[M]]K$ returns a $\lambda\mathcal{S}_1$ -term – in coercion-passing style – that applies K to the value of M .

Value translation $\Psi(V)$ is straightforward: every function $\lambda x. M$ is translated to a $\lambda\mathcal{S}_1$ -abstraction that takes as the second argument κ a coercion which is to be applied to the return value. So, the body is translated by term translation $\mathcal{K}[[M]]\kappa$.

Type translation

$$\boxed{\Psi(A) = A'}$$

$$\Psi(\star) = \star \quad \Psi(\iota) = \iota \quad \Psi(A \rightarrow B) = \Psi(A) \Rightarrow \Psi(B)$$

Coercion translation

$$\boxed{\Psi(s) = s'}$$

Value translation

$$\boxed{\Psi(V) = V'}$$

$$\begin{aligned} \Psi(\text{id}_A) &= \text{id}_{\Psi(A)} \\ \Psi(g; G!) &= \Psi(g); \Psi(G)! \\ \Psi(G^{?^p}; i) &= \Psi(G)^{?^p}; \Psi(i) \\ \Psi(s \rightarrow t) &= \Psi(s) \Rightarrow \Psi(t) \\ \Psi(\perp^{GpH}) &= \perp^{GpH} \end{aligned}$$

$$\begin{aligned} \Psi(x) &= x \\ \Psi(a) &= a \\ \Psi(\lambda x. M) &= \lambda(x, \kappa). (\mathcal{K} \llbracket M \rrbracket \kappa) \\ \Psi(U \langle\langle d \rangle\rangle) &= \Psi(U) \langle\langle \Psi(d) \rangle\rangle \end{aligned}$$

Term translation

$$\boxed{\mathcal{C} \llbracket M \rrbracket = M'}$$

$$\boxed{\mathcal{K} \llbracket M \rrbracket K = M'}$$

$$\begin{aligned} \mathcal{C} \llbracket V \rrbracket &= \Psi(V) && \text{TRC-VAL} \\ \mathcal{C} \llbracket M \langle s \rangle \rrbracket &= \mathcal{K} \llbracket M \rrbracket \Psi(s) && \text{TRC-CRC} \\ \mathcal{C} \llbracket M^A \rrbracket &= \mathcal{K} \llbracket M \rrbracket \text{id}_{\Psi(A)} && \text{otherwise TRC-ELSE} \\ \mathcal{K} \llbracket V \rrbracket K &= \Psi(V) \langle K \rangle && \text{TR-VAL} \\ \mathcal{K} \llbracket \text{op}(M, N) \rrbracket K &= \text{op}(\mathcal{C} \llbracket M \rrbracket, \mathcal{C} \llbracket N \rrbracket) \langle K \rangle && \text{TR-OP} \\ \mathcal{K} \llbracket M N \rrbracket K &= (\mathcal{C} \llbracket M \rrbracket) (\mathcal{C} \llbracket N \rrbracket, K) && \text{TR-APP} \\ \mathcal{K} \llbracket M \langle s \rangle \rrbracket K &= \text{let } \kappa = \Psi(s) ;; K \text{ in } (\mathcal{K} \llbracket M \rrbracket \kappa) && \text{TR-CRC} \\ \mathcal{K} \llbracket \text{blame } p \rrbracket K &= \text{blame } p && \text{TR-BLAME} \end{aligned}$$

■ **Figure 9** Translation into coercion-passing style (from $\lambda\mathcal{S}$ to $\lambda\mathcal{S}_1$).

We now describe the translation for terms. We write $\mathcal{K} \llbracket M \rrbracket K$ for the translation of $\lambda\mathcal{S}$ -term M with continuation coercion K . We first explain the basic transformation scheme given by the recursive function \mathcal{K}' defined by the following simpler rules:

$$\begin{aligned} \mathcal{K}' \llbracket V \rrbracket K &= \Psi(V) \langle K \rangle && \text{TR}'\text{-VAL} \\ \mathcal{K}' \llbracket \text{op}(M^{\iota_1}, N^{\iota_2}) \rrbracket K &= \text{op}(\mathcal{K}' \llbracket M \rrbracket \text{id}_{\iota_1}, \mathcal{K}' \llbracket N \rrbracket \text{id}_{\iota_2}) \langle K \rangle && \text{TR}'\text{-OP} \\ \mathcal{K}' \llbracket M^{A \rightarrow B} N^A \rrbracket K &= (\mathcal{K}' \llbracket M \rrbracket \text{id}_{\Psi(A \rightarrow B)}) (\mathcal{K}' \llbracket N \rrbracket \text{id}_{\Psi(A)}, K) && \text{TR}'\text{-APP} \\ \mathcal{K}' \llbracket M \langle s \rangle \rrbracket K &= \text{let } \kappa = \Psi(s) ;; K \text{ in } (\mathcal{K}' \llbracket M \rrbracket \kappa) && \text{TR}'\text{-CRC} \\ \mathcal{K}' \llbracket \text{blame } p \rrbracket K &= \text{blame } p && \text{TR}'\text{-BLAME} \end{aligned}$$

(We put a prime on \mathcal{K} to distinguish with the final version.)

The rule (TR'-VAL) applies to values V , where we apply coercion K to the result of value translation $\Psi(V)$.

The rule (TR'-OP) applies to primitive operations $\text{op}(M, N)$. We translate the arguments M and N with identity continuation coercions by $\mathcal{K}' \llbracket M \rrbracket \text{id}$ and $\mathcal{K}' \llbracket N \rrbracket \text{id}$ and pass them to the primitive operation. The given continuation coercion K is applied to the result. Translating subexpressions with id is one of the main differences from CPS transformation. While continuations in continuation-passing style capture the whole rest of computation, continuation coercions in coercion-passing style capture only the coercion applied right after the current computation. Since neither M nor N is surrounded by a coercion, they are translated with identity coercions of appropriate types. (Cases where a subexpression itself is a coercion application will be discussed shortly.) Careful readers may notice at this point that left-to-right evaluation of arguments is enforced by the semantics (or the definition of

evaluation contexts) of $\lambda\mathcal{S}$, not by the translation. In other words, the correctness of the translation relies on the fact that $\lambda\mathcal{S}$ evaluation is left-to-right and call-by-value. This is another point that is different from CPS transformation, which dismisses the distinction of call-by-name and call-by-value.

The rule (TR'-APP) applies to function applications $M N$. We translate function M and argument N with identity continuation coercions just like the case for primitive operations. We then pass the continuation coercion K as the second argument to function $\mathcal{K}'\llbracket M \rrbracket \text{id}$.

The rule (TR'-CRC) applies to coercion applications $M\langle s \rangle$. We can think of the sequential composition of $\Psi(s)$ and K as the continuation coercion for M . Thus, we first compute the composition $\Psi(s) ;; K$, bind its result to κ , and translate M with continuation κ . The let-expression is necessary to compose $\Psi(s)$ and K before evaluating $\mathcal{K}'\llbracket M \rrbracket \kappa$. In general, it is not necessarily the case that $\mathcal{K}'\llbracket M \rrbracket K$ evaluates K first, so if we set $\mathcal{K}'\llbracket M\langle s \rangle \rrbracket K = (\mathcal{K}'\llbracket M \rrbracket (\Psi(s) ;; K))$, then the order of computation would change by the translation and correctness of translation would be harder to show.

Lastly, the rule (TR'-BLAME) means that continuation K is discarded for blame p .

The translation \mathcal{K}' seems acceptable but, just as naïve CPS transformation leaves administrative redexes, it leaves many applications of id , which we call *administrative coercions*. We expect M and $\mathcal{K}'\llbracket M \rrbracket K$ to “behave similarly” but administrative redexes make it hard to show such semantic correspondence. Therefore, we will optimize the translation so that administrative coercions are eliminated, similarly to CPS transformations that eliminate administrative redexes [28, 3, 45, 32, 10, 8, 33].

The bottom of Figure 9 shows the optimized translation rules. The idea to eliminate administrative coercions is close to the colon translation by Plotkin [28]: we avoid translating values with administrative coercions. So, we introduce an auxiliary translation function $\mathcal{C}\llbracket M \rrbracket$, which, if M is a value V , returns $\Psi(V)$ – without a coercion application – and, if M is a coercion application $N\langle s \rangle$, returns $\mathcal{K}\llbracket N \rrbracket \Psi(s)$ – with the trivial composition $\Psi(s) ; \text{id}$ optimized away – and returns $\mathcal{K}\llbracket M \rrbracket \text{id}$ otherwise. Translation rules for primitive operations and function applications are adapted so that they use $\mathcal{C}\llbracket M \rrbracket$ to translate subexpressions.

In other words, $\mathcal{C}\llbracket M \rrbracket$ helps us precisely distinguish between id introduced by the translation and id that was present in the original term. Whenever we introduce id as an initial coercion for the translation, we first apply $\mathcal{C}\llbracket M \rrbracket$ and then apply $\mathcal{K}\llbracket M \rrbracket \text{id}$ only if necessary. We note that $\mathcal{K}\llbracket M \rrbracket \text{id} \mapsto_{\mathcal{S}_1} \mathcal{C}\llbracket M \rrbracket$ holds. We present a few examples of the translation below:

$$\begin{aligned} \Psi(\lambda x. x + 1) &= \lambda(x, \kappa). (x + 1)\langle \kappa \rangle \\ \mathcal{K}\llbracket (\lambda x. x) 5 \rrbracket \text{id} &= (\lambda(x, \kappa). x\langle \kappa \rangle) (5, \text{id}) \\ \mathcal{K}\llbracket ((\lambda x. x) 5)\langle \text{id} \rangle \rrbracket \text{id} &= \text{let } \kappa = \text{id} ! ;; \text{id}^{?p} \text{ in } (\lambda(x, \kappa). x\langle \kappa \rangle) (5, \kappa) \end{aligned}$$

The following example shows the translation of the $\lambda\mathcal{S}$ -term in Example 3 will be the $\lambda\mathcal{S}_1$ -term in Example 8.

► **Example 13.** Let U be a $\lambda\mathcal{S}$ -term $\lambda x. (x\langle \text{id}^{?p} \rangle + 2)\langle \text{id} \rangle$.

$$\begin{aligned} \Psi(U) &= \lambda(x, \kappa). (\mathcal{K}\llbracket (x\langle \text{id}^{?p} \rangle + 2)\langle \text{id} \rangle \rrbracket \kappa) \\ &= \lambda(x, \kappa). \text{let } \kappa' = \text{id} ! ;; \kappa \text{ in } (\mathcal{K}\llbracket (x\langle \text{id}^{?p} \rangle + 2) \rrbracket \kappa') \\ &= \lambda(x, \kappa). \text{let } \kappa' = \text{id} ! ;; \kappa \text{ in } (x\langle \text{id}^{?p} \rangle + 2)\langle \kappa' \rangle \\ \mathcal{K}\llbracket ((U\langle \text{id} \rangle \rightarrow \text{id}^{?p}) 3) \rrbracket \text{id} &= (\mathcal{K}\llbracket (U\langle \text{id} \rangle \rightarrow \text{id}^{?p}) \rrbracket \text{id}) (\mathcal{K}\llbracket 3 \rrbracket \text{id}, \text{id}) \\ &= (\mathcal{K}\llbracket U \rrbracket (\text{id} \rightarrow \text{id}^{?p})) (3, \text{id}) \\ &= (\Psi(U)\langle \text{id} \rangle \Rightarrow \text{id}^{?p}) (3, \text{id}) \end{aligned}$$

4.2 Correctness of Translation

Having defined the translation, we now state its correctness properties with auxiliary lemmas. (Their proofs are in the full version.)

To begin with, the translation preserves typing. Here, we write $\Psi(\Gamma)$ for the type environment satisfying: $(x : A) \in \Gamma$ if and only if $(x : \Psi(A)) \in \Psi(\Gamma)$.

► **Theorem 14** (Translation Preserves Typing).

1. If $\Gamma \vdash_S M : A$ and $s : A \rightsquigarrow B$, then $\Psi(\Gamma) \vdash_{S_1} (\mathcal{N} \llbracket M \rrbracket \Psi(s)) : \Psi(B)$.
2. If $\Gamma \vdash_S V : A$, then $\Psi(\Gamma) \vdash_{S_1} \Psi(V) : \Psi(A)$.

As for the preservation of semantics, we will prove the following theorem that states the semantics is preserved by the translation:

► **Theorem 15** (Translation Preserves Semantics). *If $\emptyset \vdash_S M : \iota$, then (1) $M \mapsto_S^* a$ iff $\mathcal{C} \llbracket M \rrbracket \mapsto_{S_1}^* a$; (2) $M \mapsto_S^* \text{blame } p$ iff $\mathcal{C} \llbracket M \rrbracket \mapsto_{S_1}^* \text{blame } p$; and (3) $M \uparrow_S$ iff $\mathcal{C} \llbracket M \rrbracket \uparrow_{S_1}$.*

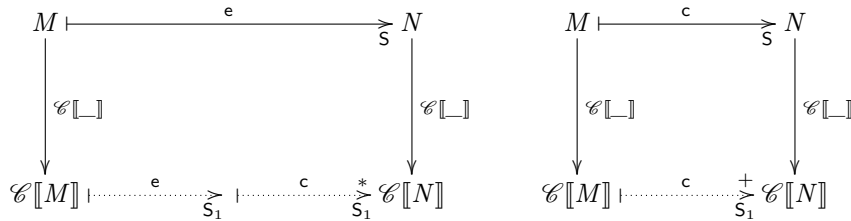
To prove this theorem, it suffices to show the left-to-right direction (Theorem 16 below) for each item because the other direction follows from Theorem 16 together with other properties: for example, if $\emptyset \vdash_S M : \iota$ and $\mathcal{C} \llbracket M \rrbracket \uparrow_{S_1}$, then M can neither get stuck (by type soundness of λS) nor terminate (as it contradicts the left-to-right direction and the fact that \mapsto_S is deterministic).

► **Theorem 16** (Translation Soundness). *Suppose $\Gamma \vdash_S M : A$. (1) If $M \mapsto_S^* V$, then $\mathcal{C} \llbracket M \rrbracket \mapsto_{S_1}^* \Psi(V)$; (2) if $M \mapsto_S^* \text{blame } p$, then $\mathcal{C} \llbracket M \rrbracket \mapsto_{S_1}^* \text{blame } p$; and (3) if $M \uparrow_S$, then $\mathcal{C} \llbracket M \rrbracket \uparrow_{S_1}$.*

A standard proof strategy would be to show that single-step evaluation in the source language is simulated by multi-step evaluation in the target language. In fact, we prove the following lemma:

► **Lemma 17** (Simulation).

1. If $M \mapsto_S^e N$, then $\mathcal{C} \llbracket M \rrbracket \mapsto_{S_1}^e \mathcal{C} \llbracket N \rrbracket$.
2. If $M \mapsto_S^c N$, then $\mathcal{C} \llbracket M \rrbracket \mapsto_{S_1}^c \mathcal{C} \llbracket N \rrbracket$.



The straightforward simulation property below follows from Lemma 17.

► **Lemma 18.** *If $M \mapsto_S N$, then $\mathcal{C} \llbracket M \rrbracket \mapsto_{S_1} \mathcal{C} \llbracket N \rrbracket$.*

As is the case for simulation proofs for CPS translation [28, 3, 45, 32, 10, 8, 33], the simulation property⁶ is quite subtle. We discuss this subtlety below.

First, it is important that the translation removes administrative identity coercions by distinguishing values and nonvalues in $\mathcal{C} \llbracket M \rrbracket$. For example, $(\lambda x. x) 5 \mapsto_S 5$ holds in λS , but the translation $\mathcal{N} \llbracket (\lambda x. x) 5 \rrbracket$ without removing administrative redexes would yield

⁶ If we had been interested only in the property that translation preserves term equivalence, we could have simplified the technical development by, say, removing the distinction between $U \langle s \rangle$ and $U \langle\langle s \rangle\rangle$. However, simulation is crucial for showing that divergence is preserved by the translation.

$((\lambda(x, \kappa). x \langle \kappa \rangle) \langle \text{id} \rangle) (5 \langle \text{id} \rangle, K)$, which performs c-evaluation before calling the function. We avoid such a situation. More formally, we prove the following lemma, which means the redex in the source is also the redex in the target.

► **Lemma 19.**

1. For any \mathcal{F} , there exists \mathcal{E}' such that for any M , $\mathcal{C}[\mathcal{F}[M]] = \mathcal{E}'[\mathcal{C}[M]]$.
2. For any \mathcal{F} and s , there exists \mathcal{E}' such that for any M , $\mathcal{C}[\mathcal{F}[M \langle s \rangle]] = \mathcal{E}'[\mathcal{C}[M] \Psi(s)]$.

To prove this lemma, the rule (TRC-CRC) also plays an important role: for example, if we removed (TRC-CRC), $\mathcal{K}[(1 + 1) \langle \text{int}! \rangle] \text{id}$ would translate to $\text{let } \kappa = \text{int}! ;; \text{id in } (1 + 1) \langle \kappa \rangle$, which performs c-evaluation before adding 1 and 1, which is the first thing the original term $(1 + 1) \langle \text{int}! \rangle$ will do.

Second, optimizing too many (identity) coercions can break simulation. We should only remove administrative identity coercions, and keep identity coercions that were present in the original term. Consider $M \stackrel{\text{def}}{=} ((\lambda x. M_1) \langle \text{id}_\iota \rightarrow \iota! \rangle) a \langle \iota?^p \rangle$ and $N \stackrel{\text{def}}{=} ((\lambda x. M_1) (a \langle \text{id}_\iota \rangle)) \langle \iota! \rangle \langle \iota?^p \rangle$, for which $M \mapsto_S N$ holds by (R-WRAP). Then,

$$\begin{aligned} \mathcal{C}[M] &= \mathcal{K}[M] \text{id} = ((\mathcal{K}[\lambda(x, \kappa). M_1] \kappa) \langle \text{id}_\iota \Rightarrow \iota! \rangle) (a, \iota?^p) \\ &\mapsto_{S_1} \text{let } \kappa' = \iota! ;; \iota?^p \text{ in } (\mathcal{K}[\lambda(x, \kappa). M_1] \kappa) (a \langle \text{id}_\iota \rangle, \kappa') = \mathcal{C}[N]. \end{aligned}$$

At one point, we defined the translation (let's call it \mathcal{K}'') so that applications of identity coercions would be removed as much as possible, namely,

$$\mathcal{K}''[N] \text{id} = \text{let } \kappa' = \iota! ;; \iota?^p \text{ in } (\mathcal{K}''[\lambda(x, \kappa). M_1] \kappa) (a, \kappa')$$

(notice that $\langle \text{id}_\iota \rangle$ on a is removed). Although $\mathcal{K}''[M] \text{id}$ and $\mathcal{K}''[N] \text{id}$ reduced to the same term, we did not quite have $\mathcal{K}''[M] \text{id} \mapsto^+ \mathcal{K}''[N] \text{id}$ as we had desired.

Third, the distinction between $U \langle s \rangle$ and $U \langle \langle s \rangle \rangle$ is crucial for ensuring that substitution commutes with the translation:

► **Lemma 20 (Substitution).** *If $\kappa \notin FV(M) \cup FV(V)$, then $(\mathcal{K}[M] \kappa)[x := \Psi(V), \kappa := K] = \mathcal{K}[M[x := V]] K$.*

Roughly speaking, if we identified a value $U \langle \langle s \rangle \rangle$ and an application $U \langle s \rangle$ of s to an uncoerced value U , then the term $U \langle s \rangle \langle t \rangle$ would allow two interpretations: an application of t to a value $U \langle s \rangle$ and applications of s and t to U and committing to either interpretation would break Lemma 20.

5 Implementation and Evaluation

5.1 Implementation

We have implemented the coercion-passing translation described in Section 4 and the semantics of λS_1 for Grift [24]⁷, an experimental compiler for gradually typed languages. GTLC+, the language that the Grift compiler implements, supports integers, floating-point numbers, Booleans, higher-order functions, local binding by **let**, (mutually) recursive definitions by **letrec**, conditional expressions, iterations, sequencing, mutable references, and vectors (mutable arrays).

⁷ The semantics of coercions in Grift is so-called D [35], which is slightly different from that of λS_1 , which is UD. Since the main difference is in the coercion composition, our technique can be applied to Grift.

The Grift compiler compiles a GTLC+ program into the C language where coercions are represented as values of a `struct` type, and operations such as coercion application and coercion composition are C functions. The compiler supports different run-time check schemes, those based on type-based casts [36] and space-efficient coercions [37]. Note that, although meta-level composition $s_1 \circ s_2$ is implemented, only nested coercions on *values* are composed; in other words, (R-MERGE) was not implemented. Thus, implicit run-time checks may break tail calls and seemingly tail-recursive functions may cause stack overflow.

We modify the compiler phases for run-time checking based on the space-efficient coercions. After typechecking a user program, the compiler inserts type-based casts to the program and converts type-based casts to space-efficient coercions, following the translation from blame calculus λB to λS [37]. Our implementation performs the coercion-passing translation after the translation into λS . It is straightforward to extend the translation scheme to language features that are not present in λS . For example, here is translation for conditional expressions:

$$\mathcal{K}[\text{if } M \text{ then } N_1 \text{ else } N_2]K = \text{if } \mathcal{C}[\![M]\!] \text{ then } (\mathcal{K}[\![N_1]\!]K) \text{ else } (\mathcal{K}[\![N_2]\!]K).$$

Since coercions are represented as `structs`, we did not have to do anything special to make coercions first-class. We modify another compiler phase that generates operations on coercions such as $M \circ N$ and (R-WRAP). The current implementation, which generates C code and uses `clang`⁸ for compilation to machine code, relies on the C compiler to perform tail-call optimizations. We have found the original compiler’s handling of recursive types hampers tail-call optimizations,⁹ so our implementation does not deal with recursive types. We leave their implementation for future work.

5.2 Even and Odd Functions

We first inspected the tail-recursive even–odd functions in GTLC+:

```
(letrec ([even (lambda ([n : A1]) : A3
  (if (= 0 n) #t (odd (- n 1))))]
 [odd (lambda ([n : A2]) : A4
  (if (= 0 n) #f (even (- n 1))))])
 (odd n))
```

where A_1 and A_2 are either `Int` or `Dyn`, and A_3 and A_4 are either `Bool` or `Dyn`. We run this program with the original and modified compilers for all combinations of A_1, A_2, A_3 , and A_4 . We call the program compiled by the original compiler `Base`, the program compiled by the modified compiler `CrcPS`.

We have confirmed that, as n increases, 12 of 16 configurations of `Base` cause stack overflow.¹⁰ In the four configurations that survived, both A_3 and A_4 are set to `Bool`. `CrcPS` never causes stack overflow for any configuration.

Although we expected that `Base` would crash if A_3 and A_4 are different, it is our surprise that `Base` causes stack overflow even when $A_3 = A_4 = \text{Dyn}$. We have found that it is due to the typing rule of Grift for conditional expressions. In Grift, if one of the branches is given a

⁸ <https://clang.llvm.org/>

⁹ The C function to compose coercions takes a pointer to a *stack-allocated* object as an argument and writes into the object when recursive coercions are composed. Although those stack-allocated objects never escape and tail-call optimization is safe, the C compiler is not powerful enough to see it.

¹⁰ The size of the run-time stack is 8 MB.

static type, say `Bool`, and the other is `Dyn`, the whole `if`-expression is given the static type and the compiler put a cast from `Dyn` on the branch of type `Dyn`. If both A_3 and A_4 are `Dyn`, the recursive calls in the two else-branches will involve casts `bool?p` from `Dyn` to `Bool` because the two then-branches are Boolean constants and the `if`-expressions are given type `Bool`. However, since the return types are declared to be `Dyn`, the whole `if`-expressions are cast back to `Dyn`, inserting injections `bool!`. Thus, every recursive call involves a projection immediately followed by an injection, as shown below, eventually causing stack overflow.

```
(letrec ([even (lambda ([n : Dyn]) : Dyn
  (if (= 0 n<int?p1) #t
      (odd (- n<int?p2 1))<bool?p3><bool!>)]
 [odd (lambda ([n : Dyn]) : Dyn
  (if (= 0 n<int?p4) #f
      (even (- n<int?p5 1))<bool?p6><bool!>))])
  (odd n))
```

5.3 Evaluation

We have conducted some experiments to measure the overhead of the coercion-passing style translation. The benchmark programs we have used are taken from Kuhlenschmidt et al. [24]¹¹; we excluded the sieve program because of the use of recursive types. We also include the even/odd program only for reference, which is relatively small compared to other programs.

We compare the running time of a benchmark program between Base and CrcPS. To take many partially typed configurations for each benchmark program into account, we focus on the so-called *fine-grained* approach, where everywhere a type is required is given either the dynamic type `Dyn` or an appropriate static type.¹² In the fine-grained approach, the number of configurations is 2^n where n is the number of type annotations. When this number is very large, we consider uniformly sampled configurations. We use the sampling algorithm¹³ from [24].

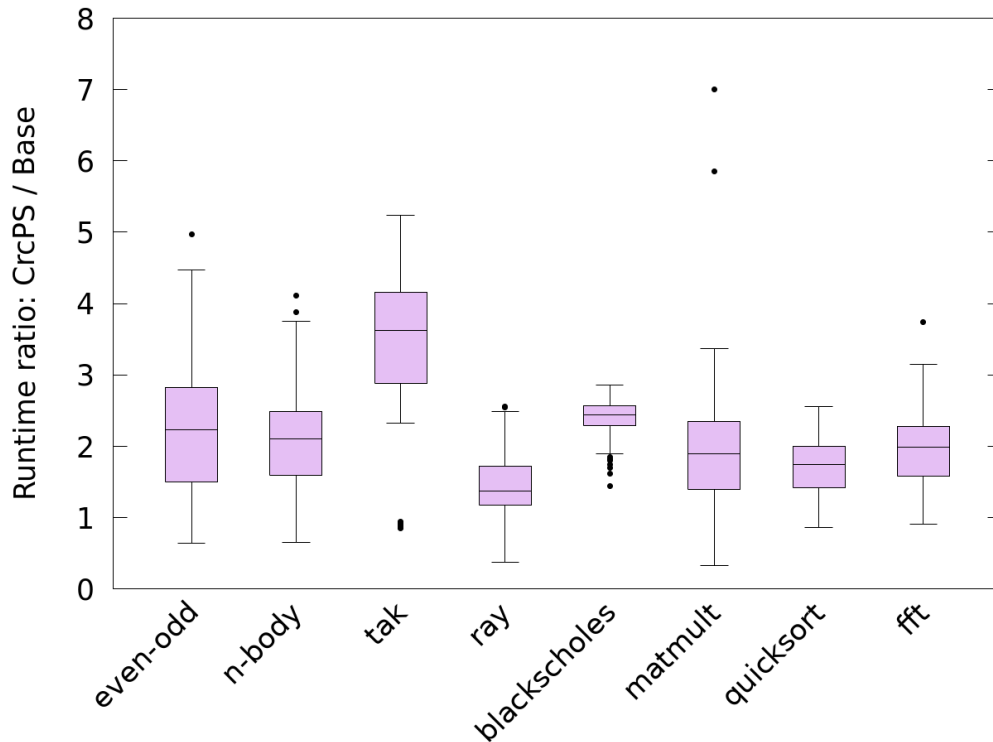
We describe the (sampled) number of partially typed configurations and main language features used for each benchmark program below. (Each benchmark program has one additional type annotation for the return type of the 0-ary main function.) For more detailed description of benchmark programs, we refer readers to Kuhlenschmidt et al. [24].

name	# of configurations	description
even-odd	all $32 = 2^5$	mutually tail-recursive functions
n-body	300 out of 2^{136}	vectors
tak	all $256 = 2^8$	recursive function
ray	300 out of 2^{280}	tuples and iterations
blackscholes	300 out of 2^{128}	vectors and iterations
matmult	300 out of 2^{33}	vectors and iterations
quicksort	300 out of 2^{44}	vectors
fft	300 out of 2^{67}	vectors

¹¹ <https://github.com/Gradual-Typing/benchmarks>

¹² The other approach is called *coarse-grained*, where functions in each module are all statically or all dynamically typed.

¹³ <https://github.com/Gradual-Typing/Dynamizer>



■ **Figure 10** A box plot for the running time ratios of CrcPS to Base across (sampled) partially typed configurations of the benchmark programs. (As is standard, the lower/upper end of a box indicates the first/third quartile, respectively, and the middle line in a box indicates the median. The length of each whisker is below 1.5 times of interquartile range, and outliers are plotted individually.)

Our benchmark method is as follows: For each partially typed configuration of a benchmark program, we measure its running time by taking the average of 5 runs for Base and CrcPS, and compute the ratio of CrcPS to Base. We use a machine with a 8-core 3.6 GHz Intel Core i7-7700 and 16 GB memory, and run the benchmark programs within a Docker container (Docker version 19.03.5) which runs Arch Linux. The generated C code is compiled by clang version 9.0.0 with `-O3` so that tail-call optimization is applied. The size of the run-time stack is set as unlimited.

Figure 10 shows the result in box plots. (Detailed plots for each benchmark are shown in the full version.) It shows that, except for `tak` (and `even-odd`), practical programs in CrcPS run up to three times as slow as Base, for most configurations. It is natural because coercion-passing style translation adds an extra coercion argument to each function. In fact, `tak` and `even-odd`, which have a lot of function calls, have large overhead compared with other programs. In `even-odd`, CrcPS performs many coercion composition operations (and one coercion application) while Base performs many coercion applications (without any coercion composition).¹⁴ Thus, the difference between Base and CrcPS for `even-odd` is partially due to the difference of the cost of coercion application and coercion composition.

¹⁴ An application of a projection coercion to an injected value is always computed by coercion composition in CrcPS, while the implementation of Base is slightly optimized for first-order types.

The benchmark programs other than `tak` and even-odd mainly concern vectors and iterations over them. Vector operations are treated in the translation as primitive operations, which we consider do not have much overhead by the translation. In fact, our translation implementation optimizes the rule (TR-OP) when its continuation is `id`: $\mathcal{K}[\![op(M, N)]\!]id = op(\mathcal{C}[\![M]\!], \mathcal{C}[\![N]\!])$ without an application of an identity coercion.

There are several configurations in which CrcPS is faster than Base but we have not figured out why this is the case.

6 Related Work

6.1 Space-Efficient Coercion/Cast Calculi

As we have already mentioned, it is fairly well known that coercions [18] and casts [43] hamper tail-call optimization and make the space complexity of the execution of a program worse than the execution under an unchecked semantics. We discuss below a few pieces of work [19, 20, 35, 38, 14, 37] addressing the problem.

To the best of our knowledge, Herman et al. [19, 20] were the first to observe the space-efficiency problem of inserted dynamic checks. They developed a variant of Henglein’s coercion calculus with semantics such that a sequence of coercion applications is eagerly composed to reduce the size of coercions. However, they identified two coercions $(c_1; c_2); c_3$ and $c_1; (c_2; c_3)$ (note that $c_1; c_2$ is not a meta-level operator but only a *formal* composition constructor); thus, an algorithm for computing coercion composition was not very clear. They did not take blame tracking [13] into account, either.

Later, Siek et al. [35] extended Herman et al. [19, 20] with a few different blame tracking strategies. The issue of identifying $(c_1; c_2); c_3$ and $c_1; (c_2; c_3)$ remained. According to their terminology, our work, which follows previous work [37], adopts the UD semantics, which allows only $\star \rightarrow \star$ as a tag to functional values, as opposed to the D semantics, which allows any function types to be used as a tag.

Siek and Wadler [38] introduced threesomes to a blame calculus as another solution to the space-efficiency problem. Threesome casts have a third type (called a mediating type) in addition to the source and target types; a threesome cast is considered a downcast from the source type to the mediating, followed by an upcast from the mediating type to the target. Threesome casts allow a simple recursive algorithm to compose two threesome casts but blame tracking is rather complicated.

Garcia [14] gave a translation from coercion calculi to threesome calculi and show that the two solutions to the space-efficiency problem are equivalent in some sense. He introduced supercoercions and a recursive algorithm to compute composition of supercoercions but they were complex, too.

Siek et al. [37] proposed yet another space-efficient coercion calculus λS , in which they succeeded in developing a simple recursive algorithm for coercion composition by restricting coercions to be in certain canonical forms – what they call space-efficient coercions. They also gave a translation from blame calculus λB to λS (via Henglein’s coercion calculus λC) and showed that the translation is fully abstract. As we have discussed already, our λS has introduced syntax that distinguishes an application $U\langle s \rangle$ of a coercion to (uncoerced) values from $U\langle\langle d \rangle\rangle$ for a value wrapped by a delayed coercion. Such distinction, which can be seen in some blame calculi [43], is not just an aesthetic choice but crucial for proving correctness of the translation.

All the above-mentioned calculi adopt a nonstandard reduction rule to compose coercions or casts even before the subject evaluates to a value, together with a nonstandard form of evaluation contexts, and as a result it has not been clear how to implement them

efficiently. Herman et al. [19, 20] sketched a few possible implementation strategies, including coercion passing, but details were not discussed. Siek and Garcia [34] showed an interpreter which performs coercion composition at tail calls. Although not showing correctness of the interpreter, their interpreter would give a hint to direct low-level implementation of space-efficient coercions. Our work addresses the problem of the nonstandard semantics in a different way – by translating a program into coercion-passing style. The difference, however, may not be so large as it may appear at first: in Siek and Garcia [34], a state of the abstract machine includes an evaluation context, which contains the information on a coercion to be applied to a return value and such a coercion roughly corresponds to our continuation coercions. More detailed analysis of the relationship between the two implementation schemes is left for future work.

Kuhlenschmidt et al. [24] built an experimental compiler Grift for gradual typing with structural types. It supports run-time checking with the space-efficient coercions of λS but does not support composition of coercions at tail positions. We have implemented our coercion-passing translation for the Grift compiler.

Greenberg [15] has studied the same space-efficiency problem in the context of manifest contract calculi [23, 16, 17] and proposed a few semantics for composing casts that involve contract checking. Feltey et al. [12] recently implemented Greenberg’s eidetic contracts on top of Typed Racket [41] but, similarly to Kuhlenschmidt et al. [24], composition is limited to a sequence of contracts applied to values.

There is other recent work for making gradual typing efficient [4, 27, 31, 29] but as far as we know, none of them addresses the problem caused by run-time checking applied to tail positions. Additionally, Castagna et al. [5] implemented a virtual machine for space-efficient gradual typing in presence of set-theoretic types, but without blame tracking. They address the problem caused by casts applied to tail positions by an approach similar to the one in the interpreter by Siek and Garcia [34]. They implemented their virtual machine and evaluated their implementation by benchmarks such as the even–odd functions.

6.2 Continuation-Passing Style

Our coercion-passing style translation is inspired by continuation-passing style translation, first formalized by Plotkin [28]. However, coercions represent only a part of the rest of computation and are, in this sense, closer to delimited continuations [7]. Roughly speaking, translating a subexpression with `id` corresponds to the `reset` operation [7] to delimit continuations. Unlike (delimited) continuations, which are usually expressed by first-class functions, coercions have compact representations and compactness can be preserved by composition.

Wallach and Felten [44] proposed security-passing style to implement Java stack inspection [25]. The idea is indeed similar to ours: each function is augmented by an additional argument to pass information on run-time security checking.

In CPS, it is crucial to eliminate administrative redexes to achieve a simulation property [28, 3, 45, 32, 10, 8, 33], which says that a reduction in the source is simulated by a sequence of (one-directional) reductions in the translation. Simulation is usually achieved by applying different translations to an application $M N$, depending on whether M and N are values or not. In addition to such value/nonvalue distinction, our coercion-passing style translation also relies on whether subterms are coercion applications or not.

Continuation-passing style eliminates the difference between call-by-name and call-by-value but our coercion-passing style translation works only under the call-by-value semantics of the target language because coercions have to be eagerly composed. It would be interesting to investigate call-by-name for either the source or the target language, or both.

6.3 First-Class Coercions

The idea of first-class coercions is also found in Cretin and Rémy [6]. Their language F_L is equipped with abstraction over coercions. However, their coercions are not for gradual typing but for parametric polymorphism and subtyping polymorphism.

7 Conclusion

We have developed a new coercion calculus λS_1 with first-class coercions as a target language of coercion-passing style translation from λS , an existing space-efficient coercion calculus. We have proved the translation preserves both typing and semantics. To achieve a simulation property, it is important to reduce administrative coercions, just as in CPS transformations. Our coercion-passing style translation solves the difficulty in implementing the semantics of λS in a faithful manner and, with the help of first-class coercions, makes it possible to implement in a compiler for a call-by-value language. We have modified an existing compiler for a gradually typed language and conducted some experiments. We have confirmed that our implementation successfully overcomes stack overflow caused by coercions at tail positions, which Kuhlenschmidt et al. [24] did not support. Our experiment has shown that for practical programs (without heavy use of function calls), the coercion-passing style translation causes slowdown up to 3 times for most partially typed configurations.

Aside from completing the implementation by adding recursive types, which the original Grift compiler supports, more efficient implementation is an obvious direction of future work. Our coercion-passing style translation introduces several identity coercions and optimizing operations on coercions will be necessary.

From a theoretical point of view, it would be interesting to extend the technique to gradual typing in the presence of parametric polymorphism [1, 2, 21, 47, 42], for which a polymorphic coercion calculus has to be studied first – Luo [26] and Kießling and Luo [22], who study coercive subtyping in polymorphic settings, may be relevant. The present design of λS_1 is geared towards coercion-passing style. For example, in λS_1 , trivial (namely identity) coercions for coercion types $A \rightsquigarrow B$ are allowed; passing coercions to dynamically typed code is prohibited; variables cannot appear as an argument to coercion constructors, like $x \Rightarrow s$. It may be interesting to study more general first-class coercions without such restrictions.

References

- 1 Amal Ahmed, Robert Bruce Findler, Jeremy G. Siek, and Philip Wadler. Blame for all. In *Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2011, Austin, TX, USA, January 26-28, 2011*, pages 201–214, 2011. doi: 10.1145/1926385.1926409.
- 2 Amal Ahmed, Dustin Jamner, Jeremy G. Siek, and Philip Wadler. Theorems for free for free: parametricity, with and without types. *PACMPL*, 1(ICFP):39:1–39:28, 2017. doi: 10.1145/3110283.
- 3 Andrew W. Appel. *Compiling with Continuations*. Cambridge University Press, 1992.
- 4 Spenser Bauman, Carl Friedrich Bolz-Tereick, Jeremy G. Siek, and Sam Tobin-Hochstadt. Sound gradual typing: only mostly dead. *PACMPL*, 1(OOPSLA):54:1–54:24, 2017. doi: 10.1145/3133878.
- 5 Giuseppe Castagna, Guillaume Duboc, Victor Lanvin, and Jeremy G. Siek. A space-efficient call-by-value virtual machine for gradual set-theoretic types. In *Proceedings of the 31st Symposium on Implementation and Application of Functional Languages, IFL 2019, Singapore, September 25-27, 2019*, 2019.

- 6 Julien Cretin and Didier Rémy. On the power of coercion abstraction. In *Proceedings of the 39th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2012, Philadelphia, Pennsylvania, USA, January 22-28, 2012*, pages 361–372, 2012. doi:10.1145/2103656.2103699.
- 7 Olivier Danvy and Andrzej Filinski. Abstracting control. In *LISP and Functional Programming*, pages 151–160, 1990. doi:10.1145/91556.91622.
- 8 Olivier Danvy and Andrzej Filinski. Representing control: A study of the CPS transformation. *Mathematical Structures in Computer Science*, 2(4):361–391, 1992. doi:10.1017/S0960129500001535.
- 9 Olivier Danvy and Lasse R. Nielsen. Syntactic theories in practice. *Electr. Notes Theor. Comput. Sci.*, 59(4):358–374, 2001. doi:10.1016/S1571-0661(04)00297-X.
- 10 Olivier Danvy and Lasse R. Nielsen. A first-order one-pass CPS transformation. *Theor. Comput. Sci.*, 308(1-3):239–257, 2003. doi:10.1016/S0304-3975(02)00733-8.
- 11 Matthias Felleisen, Mitchell Wand, Daniel P. Friedman, and Bruce F. Duba. Abstract continuations: A mathematical semantics for handling full jumps. In *LISP and Functional Programming*, pages 52–62, 1988. doi:10.1145/62678.62684.
- 12 Daniel Feltey, Ben Greenman, Christophe Scholliers, Robert Bruce Findler, and Vincent St-Amour. Collapsible contracts: fixing a pathology of gradual typing. *PACMPL*, 2(OOPSLA):133:1–133:27, 2018. doi:10.1145/3276503.
- 13 Robert Bruce Findler and Matthias Felleisen. Contracts for higher-order functions. In *Proceedings of the Seventh ACM SIGPLAN International Conference on Functional Programming (ICFP '02), Pittsburgh, Pennsylvania, USA, October 4-6, 2002.*, pages 48–59, 2002. doi:10.1145/581478.581484.
- 14 Ronald Garcia. Calculating threesomes, with blame. In *ACM SIGPLAN International Conference on Functional Programming, ICFP'13, Boston, MA, USA - September 25 - 27, 2013*, pages 417–428, 2013. doi:10.1145/2500365.2500603.
- 15 Michael Greenberg. Space-efficient manifest contracts. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, Mumbai, India, January 15-17, 2015*, pages 181–194, 2015. doi:10.1145/2676726.2676967.
- 16 Michael Greenberg, Benjamin C. Pierce, and Stephanie Weirich. Contracts made manifest. In *Proceedings of the 37th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2010, Madrid, Spain, January 17-23, 2010*, pages 353–364, 2010. doi:10.1145/1706299.1706341.
- 17 Michael Greenberg, Benjamin C. Pierce, and Stephanie Weirich. Contracts made manifest. *J. Funct. Program.*, 22(3):225–274, 2012. doi:10.1017/S0956796812000135.
- 18 Fritz Henglein. Dynamic typing: Syntax and proof theory. *Sci. Comput. Program.*, 22(3):197–230, 1994. doi:10.1016/0167-6423(94)00004-2.
- 19 David Herman, Aaron Tomb, and Cormac Flanagan. Space-efficient gradual typing. In *Proceedings of the Eighth Symposium on Trends in Functional Programming, TFP 2007, New York City, New York, USA, April 2-4, 2007.*, pages 1–18, 2007.
- 20 David Herman, Aaron Tomb, and Cormac Flanagan. Space-efficient gradual typing. *Higher-Order and Symbolic Computation*, 23(2):167–189, 2010. doi:10.1007/s10990-011-9066-z.
- 21 Yuu Igarashi, Taro Sekiyama, and Atsushi Igarashi. On polymorphic gradual typing. *PACMPL*, 1(ICFP):40:1–40:29, 2017. doi:10.1145/3110284.
- 22 Robert Kießling and Zhaohui Luo. Coercions in Hindley–Milner systems. In *Types for Proofs and Programs, International Workshop, TYPES 2003, Torino, Italy, April 30 - May 4, 2003, Revised Selected Papers*, pages 259–275, 2003. doi:10.1007/978-3-540-24849-1_17.
- 23 Kenneth Knowles and Cormac Flanagan. Hybrid type checking. *ACM Trans. Program. Lang. Syst.*, 32(2):6:1–6:34, 2010. doi:10.1145/1667048.1667051.
- 24 Andre Kuhlenschmidt, Deyaaeldeen Almahallawi, and Jeremy G. Siek. Toward efficient gradual typing for structural types via coercions. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2019, Phoenix, AZ, USA, June 22-26, 2019*, pages 517–532, 2019. doi:10.1145/3314221.3314627.

- 25 Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, 2nd edition, 1999.
- 26 Zhaohui Luo. Coercions in a polymorphic type system. *Mathematical Structures in Computer Science*, 18(4):729–751, 2008. doi:10.1017/S0960129508006804.
- 27 Fabian Muehlboeck and Ross Tate. Sound gradual typing is nominally alive and well. *PACMPL*, 1(OOPSLA):56:1–56:30, 2017. doi:10.1145/3133880.
- 28 Gordon D. Plotkin. Call-by-name, call-by-value and the λ -calculus. *Theor. Comput. Sci.*, 1(2):125–159, 1975. doi:10.1016/0304-3975(75)90017-1.
- 29 Aseem Rastogi, Nikhil Swamy, Cédric Fournet, Gavin M. Bierman, and Panagiotis Vekris. Safe & efficient gradual typing for TypeScript. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, Mumbai, India, January 15-17, 2015*, pages 167–180, 2015. doi:10.1145/2676726.2676971.
- 30 John C. Reynolds. Definitional interpreters for higher-order programming languages. *Higher-Order and Symbolic Computation*, 11(4):363–397, December 1998. This paper originally appeared in the Proceedings of the ACM National Conference, volume 2, August 1972, ACM, New York, pages 717–740.
- 31 Gregor Richards, Ellen Arteca, and Alexi Turcotte. The VM already knew that: leveraging compile-time knowledge to optimize gradual typing. *PACMPL*, 1(OOPSLA):55:1–55:27, 2017. doi:10.1145/3133879.
- 32 Amr Sabry and Matthias Felleisen. Reasoning about programs in continuation-passing style. *Lisp and Symbolic Computation*, 6(3-4):289–360, 1993.
- 33 Amr Sabry and Philip Wadler. A reflection on call-by-value. *ACM Trans. Program. Lang. Syst.*, 19(6):916–941, 1997. doi:10.1145/267959.269968.
- 34 Jeremy G. Siek and Ronald Garcia. Interpretations of the gradually-typed lambda calculus. In *Proceedings of the 2012 Annual Workshop on Scheme and Functional Programming, Scheme 2012, Copenhagen, Denmark, September 9-15, 2012*, pages 68–80, 2012. doi:10.1145/2661103.2661112.
- 35 Jeremy G. Siek, Ronald Garcia, and Walid Taha. Exploring the design space of higher-order casts. In *Programming Languages and Systems, 18th European Symposium on Programming, ESOP 2009, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009, York, UK, March 22-29, 2009. Proceedings*, pages 17–31, 2009. doi:10.1007/978-3-642-00590-9_2.
- 36 Jeremy G. Siek and Walid Taha. Gradual typing for functional languages. In *Scheme and Functional Programming Workshop*, pages 81–92, 2006.
- 37 Jeremy G. Siek, Peter Thiemann, and Philip Wadler. Blame and coercion: together again for the first time. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15-17, 2015*, pages 425–435, 2015. doi:10.1145/2737924.2737968.
- 38 Jeremy G. Siek and Philip Wadler. Threesomes, with and without blame. In *Proceedings of the 37th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2010, Madrid, Spain, January 17-23, 2010*, pages 365–376, 2010. doi:10.1145/1706299.1706342.
- 39 Asumu Takikawa, Daniel Feltey, Ben Greenman, Max S. New, Jan Vitek, and Matthias Felleisen. Is sound gradual typing dead? 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*, pages 456–468, 2016. doi:10.1145/2837614.2837630.
- 40 Sam Tobin-Hochstadt and Matthias Felleisen. Interlanguage migration: from scripts to programs. In *Proc. of Dynamic Languages Symposium*, pages 964–974, 2006. doi:10.1145/1176617.1176755.
- 41 Sam Tobin-Hochstadt and Matthias Felleisen. The design and implementation of Typed Scheme. In *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, 2008. doi:10.1145/1328438.1328486.

- 42 Matías Toro, Elizabeth Labrada, and Éric Tanter. Gradual parametricity, revisited. *PACMPL*, 3(POPL):17:1–17:30, 2019. doi:10.1145/3290330.
- 43 Philip Wadler and Robert Bruce Findler. Well-typed programs can't be blamed. In *Programming Languages and Systems, 18th European Symposium on Programming, ESOP 2009, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009, York, UK, March 22-29, 2009. Proceedings*, pages 1–16, 2009. doi:10.1007/978-3-642-00590-9_1.
- 44 Dan S. Wallach and Edward W. Felten. Understanding java stack inspection. In *Security and Privacy - 1998 IEEE Symposium on Security and Privacy, Oakland, CA, USA, May 3-6, 1998, Proceedings*, pages 52–63, 1998. doi:10.1109/SECPRI.1998.674823.
- 45 Mitchell Wand. Correctness of procedure representations in higher-order assembly language. In *Mathematical Foundations of Programming Semantics, 7th International Conference, Pittsburgh, PA, USA, March 25-28, 1991, Proceedings*, pages 294–311, 1991. doi:10.1007/3-540-55511-0_15.
- 46 Andrew K. Wright and Matthias Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115(1):38–94, November 1994.
- 47 Ningning Xie, Xuan Bi, and Bruno C. d. S. Oliveira. Consistent subtyping for all. In *Programming Languages and Systems - 27th European Symposium on Programming, ESOP 2018, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2018, Thessaloniki, Greece, April 14-20, 2018, Proceedings*, pages 3–30, 2018. doi:10.1007/978-3-319-89884-1_1.

Multiparty Session Programming With Global Protocol Combinators

Keigo Imai¹ 

Gifu University, Japan

keigo@gifu-u.ac.jp

Rumyana Neykova 

Brunel University London, United Kingdom

Rumyana.Neykova@brunel.ac.uk

Nobuko Yoshida 

Imperial College London, United Kingdom

n.yoshida@imperial.ac.uk

Shoji Yuen 

Nagoya University, Japan

yuen@i.nagoya-u.ac.jp

Abstract

Multiparty Session Types (MPST) is a typing discipline for communication protocols. It ensures the absence of communication errors and deadlocks for well-typed communicating processes. The state-of-the-art implementations of the MPST theory rely on (1) *runtime linearity checks* to ensure correct usage of communication channels and (2) external domain-specific languages for specifying and verifying multiparty protocols.

To overcome these limitations, we propose a library for programming with *global combinators* – a set of functions for writing and verifying multiparty protocols in OCaml. Local behaviours for *all* processes in a protocol are inferred *at once* from a global combinator. We formalise global combinators and prove a sound realisability of global combinators – a well-typed global combinator derives a set of local types, by which typed endpoint programs can ensure type and communication safety. Our approach enables fully-static verification and implementation of the whole protocol, from the protocol specification to the process implementations, to happen in the same language.

We compare our implementation to untyped and continuation-passing style implementations, and demonstrate its expressiveness by implementing a plethora of protocols. We show our library can interoperate with existing libraries and services, implementing DNS (Domain Name Service) protocol and the OAuth (Open Authentication) protocol.

2012 ACM Subject Classification Software and its engineering → Concurrent programming structures; Theory of computation → Type structures; Software and its engineering → Functional languages; Software and its engineering → Polymorphism

Keywords and phrases Multiparty Session Types, Communication Protocol, Concurrent and Distributed Programming, OCaml

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2020.9

Related Version A full version of the paper [25] is available at <http://arxiv.org/abs/2005.06333>.

Supplementary Material ECOOP 2020 Artifact Evaluation approved artifact available at <https://doi.org/10.4230/DARTS.6.2.18>.

A source code repository for the accompanying artifact is available at <https://github.com/keigo/ocaml-mpst/>.

¹ Corresponding author



Funding Our work is partially supported by the first author’s visitor funding to Imperial College London and Brunel University London supported by Gifu University, VeTSS, JSPS KAKENHI Grant Numbers JP17H01722, JP17K19969 and JP17K12662, JSPS Short-term Visiting Fellowship S19068, EPSRC Doctoral Prize Fellowship, and EPSRC EP/K011715/1, EP/K034413/1, EP/L00058X/1, EP/N027833/1, EP/N028201/1, EP/T006544/1 and EP/T014709/1.

Acknowledgements We thank Jacques Garrigue and Oleg Kiselyov for their comments on an early version of this paper.

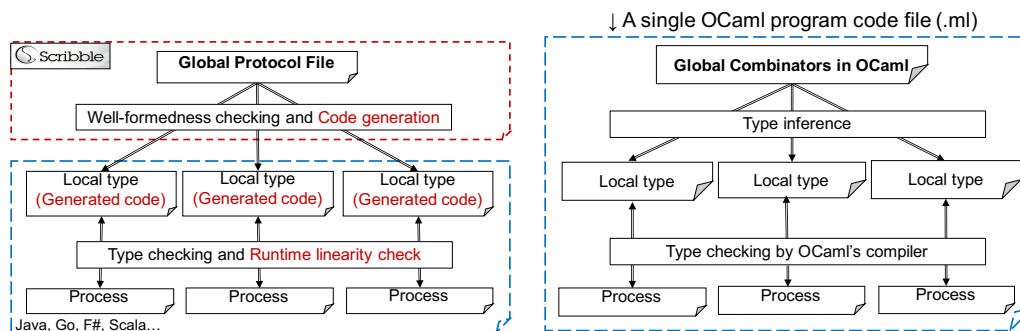
1 Introduction

Multiparty Session Types. Multiparty Session Types (MPST) [20, 11, 21] is a theoretical framework that stipulates how to write, verify and ensure correct implementations of communication protocols. The methodology of programming with MPST (depicted in Fig. 1(a)) starts from a communication protocol (a *global type*) which specifies the behaviour of a system of interacting processes. The local behaviour (a *local type*) for each endpoint process is then algorithmically *projected* from the protocol. Finally, each endpoint process is implemented in an endpoint host language and type-checked against its respective local type by a session typing system. The guarantee of session types is that a system of well-typed endpoint processes *does not go wrong*, i.e. it does not exhibit communication errors such as reception errors, orphan messages or deadlocks, and satisfies session fidelity, i.e. the local behaviour of each process follows the global specification.

The theoretical MPST framework ensures desirable safety properties. In practice, session types implementations that enforce these properties *statically*, i.e. at compile-time, are limited to binary (two party protocols) [43, 39, 31, 41]. Extending binary session types implementations to multiparty interactions, which support static linearity checks (i.e., linear usage of channels), is non-trivial, and poses two implementation challenges.

(C1) How global types can be specified and verified in a general-purpose programming language? Checking compatibility of two communicating processes relies on *duality*, i.e., when one process performs an action, the other performs a complementary (dual) action. Checking the compatibility of multiple processes is more complicated, and relies on the existence of a *well-formed* global protocol and the syntax-directed procedure of *projection*, which derives local types from a global specification. A global protocol is considered *well-formed*, if local types can be derived via projection. Since global types are far from the types of a “mainstream” programming language, state-of-the-art MPST implementations [22, 36, 47, 9] use external domain-specific protocol description languages and tools (e.g. the Scribble toolchain [50]) to specify global types and to implement the verification procedure of projection. The usage of external tools for protocol description and verification widens the gap between the specification and its implementations and makes it more difficult to locate protocol violations in the program, i.e. the correspondence between an error in the program and the protocol is less apparent.

(C2) How to implement safe multiparty communication over binary channels? The theory of MPST requires processes to communicate over multiparty channels – channels that carry messages between two or more parties; their types stipulate the precise sequencing of the communication between multiple processes. Additionally, multiparty channels has to be used linearly, i.e. exactly once. In practice, however, (1) communication channels are binary, i.e. a TCP socket for example connects only two parties, and hence its type can describe interactions between two entities only; (2) most languages do not support typing of linear resources. Existing MPST implementations [22, 36, 47, 9] apply two workarounds.



■ **Figure 1** (a) State-of-the-art MPST implementations and (b) `ocaml-mpst` methodology.

To preserve the order of interactions when implementing a multiparty protocol over binary channels, existing works use code generation (e.g. [50]) and generate local types (APIs) for several (nominal) programming languages. Note that although the interactions order is preserved, most of these implementations [22, 36, 9] still require type-casts on the underlying channels, compromising type safety of the host type system. To ensure linear usage of multiparty channels, *runtime checks* are inserted to detect if a channel has been used more than once. This is because the type systems of their respective host languages do not provide static linearity checking mechanism.

Our approach. This paper presents a library for programming MPST protocols in OCaml that solves the above challenges. Our library, `ocaml-mpst`, allows to specify, verify and implement MPST protocols in a single language, OCaml. Specifically, we address **(C1)** by developing *global combinators*, an embedded DSL (EDSL) for writing global types in OCaml. We address **(C2)** by encoding multiparty channels into *channel vectors* – a data structure, storing a nested sequence of binary channels. Moreover, `ocaml-mpst` verifies *statically* the linear usage of communication channels, using OCaml’s strong typing system and supports session delegation.

The key device in our approach is the discovery that in a system with variant and record types, checking compatibility of local types coincides with existence of least upper bound w.r.t. subtyping relation. This realisation enables a fully static MPST implementation, i.e., static checking not only on local but also on global types in a general purpose language.

Programming with `ocaml-mpst` (Fig. 1(b)) closely follows the “top-down” methodology of MPST, but differs from the traditional MPST framework in Fig. 1(a). To use our library, a programmer specifies the global protocol with a set of global combinators. The OCaml typechecker verifies correctness of the global protocol and infers local types from global combinators. A developer implements the endpoint processes using our `ocaml-mpst` API. Finally, the OCaml type checker verifies that the API is used according to the inferred type.

The benefits of `ocaml-mpst` are that it is (1) *lightweight* – it does not depend on any external code-generation mechanism, verification of global protocols is reduced to typability of global combinators; (2) *fully-static* – our embedding integrates with recent techniques for static checking of binary session types and linearly-typed lists [27, 24], which we adopt to implement multiparty session channels and session delegation; (3) *usable* – we can auto-detect and correct protocol violations in the program, guided by OCaml programming environments like Merlin [4]; (4) *extensible* – while most MPST implementations rely on a nominal typing, we embed session types in OCaml’s *structural* types, and preserve session subtyping [17]; and (5) *expressive* – we can type strictly more processes than [48] (see § 7).

```

1 let oAuth = (s -->c) login @@ (c -->a) pwd @@ (a -->s) auth @@ finish (* global protocol*)
2 (* The client process *)
3 let cliThread () =
4   let ch = get_ch c oAuth in
5   let `login(x, ch) = recv ch#role_S in
6   let ch = send ch#role_A#pwd "pass" in
7   close ch
8
9 (* The service process *)
10 let srvThread () =
11  let ch = get_ch s oAuth in
12  let ch = send ch#role_C#login "Hi" in
13  let `auth(_,ch) = recv ch#role_A in
14  close ch
15
16 (* The authenticator process *)
17 let authThread () =
18  let ch = get_ch a oAuth in
19  let `pwd(code,ch) = recv ch#role_C in
20  let ch = send ch#role_S#auth true in
21  close ch
22
23 (* start all processes *)
24 let () =
25  List.iter Thread.join [
26    Thread.create cliThread ();
27    Thread.create srvThread ();
28    Thread.create authThread ()]

```

■ **Figure 2** Global protocol and local implementations for OAuth protocol ².

Contributions. Contributions and the outline of the paper are as follows:

- § 2 gives an overview of programming with `ocaml-mpst`, a library in OCaml for specification, verification and implementations of communication protocols.
- § 3 formalises global combinators, presents their typing system, and proves a *sound realisability of global combinator*, i.e. a set of local types inferred from a global combinator can type a channel which embeds a set of endpoint behaviours as OCaml data structures.
- § 4 discusses the design and implementation of global combinators.
- § 5 summarises the `ocaml-mpst` communication library and explains how we utilise advanced features/libraries in OCaml to enable dynamic/static linearity checking on channels.
- § 6 evaluates `ocaml-mpst`. We compare `ocaml-mpst` with several different implementations and demonstrate the expressiveness of `ocaml-mpst` by showing implementations of MPST examples, as well as a variety of real-world protocols. We demonstrate our library can interoperate with existing libraries and services, namely we implement DNS (Domain Name Service) and the OAuth (Open Authentication) protocols on top of existing libraries. We discuss related work in § 7 and conclude with future work in § 8. Full proofs, omitted definitions and examples can be found in [25]. Our implementation, `ocaml-mpst` is available at <https://github.com/keigo/ocaml-mpst> including benchmark programs and results.

2 Overview of OCaml Programming with Global Combinators

This section gives an overview of multiparty session programming in `ocaml-mpst` by examples. It starts from declaration of global combinators, followed by endpoint implementations. We also demonstrate how errors can be reported by an OCaml programming environment like Merlin [4]. In the end of this section, we show the syntax of global combinators and the constructs of `ocaml-mpst` API in Fig. 5. The detailed explanation of the implementations of the constructs is deferred to § 4.

From global combinators to communication programs. We illustrate *global combinators* starting from a simple authentication protocol (based on OAuth 2.0 [18]). A full version of the protocol is implemented and discussed in § 6. Fig. 2 shows the complete OCaml implementation of the protocol, from the protocol specification (using global combinators) to the endpoint implementations (using `ocaml-mpst` API).

The protocol consists of three parties, a service **s**, a client **c**, and an authenticator **a**. The interactions between the parties (hereafter also called *roles*) proceed as follows: (1) the service **s** sends to the client **c** a `login` message containing a greeting (of type `string`); (2) the client then continues by sending its password (`pwd`) (of type `string`) to the authenticator **a**; and (3) finally the authenticator **a** notifies **s**, by sending an `auth` message (of type `bool`), whether the client access is authorised.

The global protocol `oAuth` in Line 1 is specified using two global combinators, `-->` and `finish`. The former represents a point-to-point communication between two roles, while the latter signals the end of a protocol. The operator `@@` is a right-associative function application operator to eliminate parentheses, i.e., `(c --> a) pwd @@ exp` is equivalent to `(c --> a) pwd (exp)`, where `-->` works as a four-ary function which takes roles **c** and **a** and label `pwd` and continuation `exp`. We assume that `login`, `pwd` and `auth` are predefined by the user as *label objects* with their *payload types* of `string`, `string` and `bool`, respectively³. Similarly, **s**, **c** and **a** are predefined *role objects*. We elaborate on how to define these custom labels and roles in § 4.

The execution of the `oAuth` expression returns a tuple of three *channel vectors* – one for each role in the global combinator. Each element of the tuple can be extracted using an index, encoded in role objects (**c**, **s**, and **a**). Intuitively, the role object **c** stores a functional pointer that points to the first element of the tuple, **s** points to the second, and **a** to the third element. The types of the extracted channel vectors reflect the local behaviour that each role, specified in the protocol, should implement. Channel vectors are objects that hide the *actual bare communication channels* shared between every two communicating processes.

Lines 3–21 present the implementations for all three processes specified in the global protocol. We explain the implementation for the client – `cliThread` (Lines 3–7). Other processes are similarly implemented. Line 4 extracts the channel vector that encapsulates the behaviour of the client, i.e the first element of `oAuth`. This is done by using the function `get_ch` (provided by our library) applied to the role object **c** and the expression `oAuth`.

Our library provides two main communication primitives, namely `send` and `recv`. To statically check communication structures using types, we exploit OCaml’s *structural* types of objects and polymorphic variants (rather than their nominal counterparts of records and ordinary variants). In Line 5, `ch#role_S` is an invocation of method `role_S` on an object `ch`. The `recv` primitive waits on a *bare channel* returned by the method invocation. The returned value is matched against a variant tag indicating the input label `login` with the pair of the payload value `x` and a continuation `ch` (shadowing the previous usage of `ch`). Then, on Line 6, two method calls on `ch` are performed, e.g `ch#role_A#pwd`, which extract a communication channel for sending a password (`pwd`) to the authenticator. This channel is passed to the `send` primitive, along with the payload value `"pass"`. Then, `let` rebinds the name `ch` to the continuation returned by `send` and on Line 7 the channel is closed. Each operation is guided by the host OCaml type system, via *channel vector type*. For example, the client channel `ch` extracted in Line 4 has a channel vector type (inferred by OCaml type checker) `<role_S: [login of string * t] inp>` which denote reception (suffixed by `inp`) from server of a `login` label, then continuing to `t`, where `t` is `<role_A: <pwd: (string, close) out>>` denoting sending (`out`) to authenticator of a `pwd` label, followed by closing. Note that the type `<f: t>` denotes an OCaml object with a field `f` of type `t`; `[m of t]` is a (polymorphic) variant type having a tag `m` of type `t`. Finally, in Lines 25–28 all processes are started in new threads.

² We use a simplified syntax that support the in-built communication transport of Ocaml. For the full syntax of the library that is parametric on the transport, see the repository.

³ To be precise, the labels are *polymorphic* on their payload types which are instantiated at the point

```

1 let oAuth2 () =
2   (choice_at s (to_s login_cancel)
3    (s, oAuth ())
4    (s, (s -->c) cancel @@
5     (c -->a) quit @@
6     finish))
1 let oAuth3 () =
2   fix (fun repeat ->
3     (choice_at s (to_s oauth2_retry)
4      (s, oAuth2 ()
5       (s, (s -->c) retry @@
6        repeat)))

```

(a) Protocol With Branching.

(b) Protocol With Branching & Recursion.

■ **Figure 3** Extended `oAuth` protocols.

On the expressiveness of well-typed global protocols. Fig. 3 shows two global protocols that extend `oAuth` with new behaviours. In Fig. 3a, the global combinator `choice_at` specifies a branching behaviour at role `s`. In the first case (Line 3), the protocol proceeds with protocol `oAuth`. In the second case (Line 5) the service sends `cancel`, to the client, and the client sends a `quit` message to the authenticator. The deciding role, `s`, is explicit in each branch. The choice combinator requires a user-defined `(to_s login_cancel)` (Line 2) that specifies concatenation of two objects for sending in branches. Its implementation is straightforward (see § 4). The protocol `oAuth3` in Fig. 3b reuses `oAuth2` and further elaborates its behaviour by offering a retry option. It demonstrates a recursive specification where the `fix` combinator binds the protocol itself to variable `repeat`.

The implementation of the corresponding client code for Fig. 3a is shown on Fig. 4a. The code is similar as before, but uses a pattern matching against multiple tags ``login` and ``cancel` to specify an *external choice* on the client, i.e the client can receive messages of different types and exhibit different behaviour according to received labels. The behaviour that a role can send messages of different types, which is often referred to as an *internal choice*, is represented as an object with multiple methods.

Our implementation also preserves the subtyping relation in session types [17], i.e the safe replacement of a channel of more capabilities in a context where a channel of less capabilities is expected. Session subtyping is important in practice since it ensures backward compatibility for protocols: a new version of a protocol does not break existing implementations. For example, the client function in Fig. 4a is typable under both protocols `oAuth2` and `oAuth3` since the type of the channel stipulating the behaviour for role `c` in `oAuth2` (receiving either message ``login` or ``cancel`) is a subtype of the channel for `c` in `oAuth3` (receiving ``login`, ``cancel`, or ``retry`).

Static linearity and session delegation. The implementations presented in Fig. 2, as well as Fig. 4a detect linearity violations at runtime, as common in MPST implementations [22, 47] in a non-substructural type system. We overcome this dynamic checking issue by an alternative approach, listed in Fig. 4b. We utilise an extension (`let%lin`) for linear types in OCaml [24] that statically enforces linear usage of resources by combining the usage of parameterised monads [29, 2, 40] and lenses [16]. Our library is parameterised on the chosen approach, static or dynamic. A few changes are made to avoid explicit handling of linear resources: (1) `ch` in Fig. 4b refers to a *linear* resource and has to be matched against a *linear pattern* prefixed by `#`. (2) Roles and labels are now specified as a *selector* function of the form `(fun x->x#role#label)`.

where they are used.

```

1 match recv ch#role_S with
2 | `login(pass, ch) ->
3   let ch = send ch#role_A#pwd pass
4   in close ch
5 | `cancel(_, ch) ->
6   let ch = send ch#role_A#quit ()
7   in close ch

```

(a) Dynamic Linearity Checking.

(b) Static Linearity Checking.

■ **Figure 4** Two Modes on Linearity Checking.

Global Combinators to Local Types where t_i is a local type at r_i in g ($1 \leq i \leq n$)	
Global Combinator	Synopsis
$(r_i \dashrightarrow r_j) m g$	Transmission from r_i to r_j of label m (with a payload).
<code>choice_at r_a mrg (r_a, g_1) (r_a, g_2)</code>	Branch to g_1 or g_2 guided by r_a .
<code>finish</code>	Finished session.
<code>fix (fun x -> g)</code>	Recursion. Free occurrences of x is equivalent to g itself.
Local Types and Communication Primitives	
Communication Primitive	Synopsis
<code>send s#role_r#m_k e</code>	Send to role r label m_k with payload e , returning continuation.
<code>let $\backslash m(x, s) =$ receive s#role_r</code> <code>in e</code>	Receive from r label m with payload $x : v$ and continue to e with endpoint $s : t$
<code>match receive s#role_r with</code> <code> $\backslash m_1(x_1, s) -> e_1$ \dots</code> <code> $\backslash m_n(x_n, s) -> e_n$</code>	Receive from r one of labels $\{m_i\}$ ($1 \leq i \leq n$) where payload is v_i and continue with t_i in e_i
<code>close s</code>	Closes a session

■ **Figure 5** (a) Global Combinators (top) and (b) Communication APIs of `ocaml-mpst` (bottom).

Our implementation is also the first to support *static* multiparty sessions delegation (the capability to pass a channel to another endpoint): our encoding yields it for free, via existing mechanisms for binary delegation (see § 4).

Errors in global protocol and `ocaml-mpst` endpoint programs. Our framework ensures that a well-typed `ocaml-mpst` program precisely implements the behaviour of its defined global protocol. Hence, if a program does not conform to its protocol, a compilation error is reported. Fig. 6 shows the error reported when swapping the order of send and receive actions (Lines 6 and 5) in the client implementation in Fig. 2. Similarly, errors will also be reported if we misspell any of the methods `pwd`, `role_A`, or `role_C`.

Similarly, an error is reported if the global protocol is *not safe* (which corresponds to an ill-formed MPST protocols [14]) since this may lead to *unsafe* implementations. Consider Fig. 6 (b), where we modify `oAuth2` such that `s` sends a `cancel` message to `a`. This protocol (`oAuth4`) exhibits a race condition: even if all parties adhere to the specified behaviour, `c` can send a `quit` before `s` sends `login`, which will lead to a deadlock on `s`. Our definition of global combinators prevents such ill-formed protocols, and the OCaml compiler will report an error. The actual error message reported in OCaml detects the mismatch between `a` and `c`, indicating violation of the *active role* property in the MPST literature [14] – the sender must send to the same role.

3 Formalisms and Typing for Global Combinators

This section formalises global combinators and their typing system, along a formal correspondence between a global combinator and channel vectors. The aim of this section is to provide a guidance towards descriptions of the implementations presented in § 4.5.

We first give the syntax of global combinators and channel vectors in § 3.1. We then propose a typing system of global combinators in § 3.2, illustrating that the rules check their well-formedness. We define derivation of channel vectors from global combinators in § 3.3. The main theorem (Theorem 3.11) states that a well-typed global combinator always derives a channel vector which is typable by a corresponding set of local types, i.e. any well-typed global combinator is soundly realisable by a tuple of well-typed channel vectors.

3.1 Global Combinators and Channel Vector Types

Global combinators. denote a communication protocol which describes the whole conversation scenario of a multiparty session.

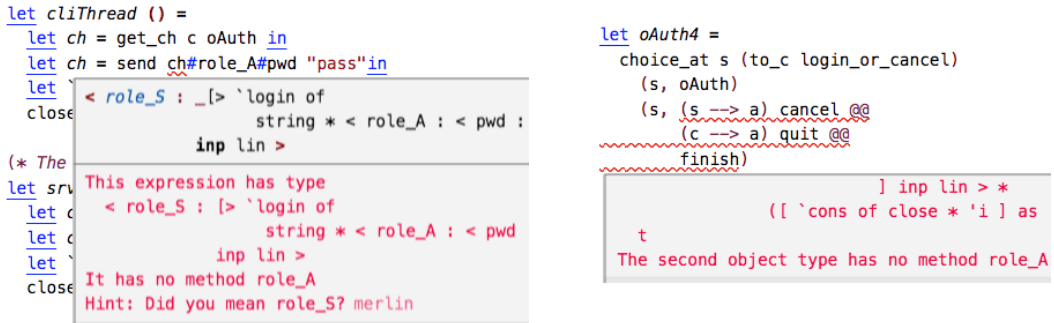
► **Definition 3.1** (Global combinators and channel vector types). The syntax of *global combinators*, written g, g', \dots , are given as:

$$g ::= (p \rightarrow q) m:T g \mid \text{choice } p \{g_i\}_{i \in I} \mid \text{fix } x \rightarrow g \mid x \mid \text{finish}$$

where the syntax of *payload types* S, T, \dots (also called *channel vector types*) is given below:

$$T, S ::= !T \mid ?T \mid \#T \mid T_1 \times \dots \times T_n \mid \langle l_i : T_i \rangle_{i \in I} \mid [l_i _T_i]_{i \in I} \mid \mu t. T \mid t \mid \bullet$$

The formal syntax of global combinators comes from Scribble [50] and corresponds to the standard global types in MPSTs [37]. We assume a set of participants ($\mathfrak{P} = \{p, q, r, \dots\}$), and that of alphabets ($\mathfrak{A} = \{\text{ok}, \text{cancel}, \dots\}$). **Communication combinator** $(p \rightarrow q) m:T g$ states that participant p can send a message of type T with label m to participant q and that the interaction described in g follows. We require $p \neq q$ to prevent self-sent messages. We omit the payload type when *unit* type \bullet , and assume T is *closed*, i.e. it does not contain free recursive variables. **Choice combinator** $\text{choice } p \{g_i\}_{i \in I}$ is a branching in a protocol where p makes a decision (i.e. an output) on which branch the participants will take. **Recursion** $\text{fix } x \rightarrow g$ is for recursive protocols, assuming that variables (x, x', \dots) are guarded in the standard way, i.e. they only occur under the communication combinator. **Termination** finish represents session termination. We write $p \in \text{roles}(g)$ (or simply $p \in g$) iff, for some q , either $p \rightarrow q$ or $q \rightarrow p$ occurs in g .



■ **Figure 6** Type Errors Reported by Visual Studio Code (Powered by Merlin), in (a) Local Type (left) and (b) Global Combinator (right).

► **Example 3.2.** The global combinator g_{Auth} below specifies a variant of an authentication protocol in Fig. 3 where $T = \text{string}$ and `client` sends `auth` to `server`, then `server` replies with either `ok` or `cancel`.

$$g_{\text{Auth}} = (\text{c} \rightarrow \text{s}) \text{auth}:T \left(\text{choices } \text{s} \{ (\text{s} \rightarrow \text{c}) \text{ok}:T \text{ finish}, (\text{s} \rightarrow \text{c}) \text{cancel}:T \text{ finish} \} \right)$$

Channel vector types. abstract behaviours of each participant using standard data structure and channels. We assume labels $\mathbf{l}, \mathbf{l}', \dots$ range over $\mathfrak{R} \cup \mathfrak{A}$. Types $!T$ and $?T$ denote *output* and *input channel types*, with a value or channel of type T (note that the syntax includes *session delegation*). $\#T$ is an *io-type* which is a subtype of both input or output types [46]. $T_1 \times \dots \times T_n$ is an n -ary *tuple type*. $\langle \mathbf{l}_i : T_i \rangle_{i \in I}$ is a *record type* where each field \mathbf{l}_i has type T_i for $i \in I$. $[\mathbf{l}_i _ T_i]_{i \in I}$ is a *variant type* [46] where each \mathbf{l}_i is a possible *tag* (or *constructor*) of that type and T_i is the argument type of the tag. In both record and variant types, we assume the fields and tags are distinct (i.e. in $\langle \mathbf{l}_i : T_i \rangle_{i \in I}$ and $[\mathbf{l}_i _ T_i]_{i \in I}$, we assume $\mathbf{l}_i \neq \mathbf{l}_j$ for all $i \neq j$). The symbol \bullet denotes a unit type. Type \mathbf{t} is a variable for recursion. A *recursive type* takes an equi-recursive viewpoint, i.e. $\mu \mathbf{t}. T$ is viewed as $T \{ \mu \mathbf{t}. T / \mathbf{t} \}$. Recursion variables are guarded and payload types are closed.

Channel vectors: Session types as record and variant types. The execution model of MPST assumes that processes communicate by exchanging messages over input/output (I/O) channels. Each channel has the capability to communicate with multiple other processes. A *local session type* prescribes the local behaviour for a role in a global protocol by assigning a type to the communication channel utilised by the role. More precisely, a local session type specifies the exact order and payload types for the communication actions performed on each channel (see Fig. 1(a)). In practice, processes communicate on a low-level *bi-directional* I/O channels (*bare channels*), which are used for synchronisation of two (but *not* multiple) processes. Therefore, to implement local session types in practice, a process should utilise multiple bare channels, preserving the order, in which such channels should be used. We encode local session types as channel vector types, which *wrap* bare channels (represented in our setting by $?T, !T, \#T$ types) in record and variant types. This is illustrated in the following table, with the corresponding local session types for reference.

Behaviour	Channel vector type	Local session type [49]
Selection (Output choice)	$\langle \mathbf{q} : \langle \mathbf{m}_i : !S_i \times T_i \rangle_{i \in I} \rangle$	$\mathbf{q} \oplus_{i \in I} \mathbf{m}_i (S_i) . T_i$
Branching (Input choice)	$\langle \mathbf{q} : ?[\mathbf{m}_i _ S_i \times T_i]_{i \in I} \rangle$	$\mathbf{q} \&_{i \in I} \mathbf{m}_i (S_i) . T_i$
Recursion	$\mu \mathbf{t}. T, \mathbf{t}$	$\mu \mathbf{t}. T, \mathbf{t}$
Closing	\bullet	end

Intuitively, the behaviour of sending a message is represented as a record type, which stores inside its fields a bare output channel and a continuation; the input channel required when receiving a message is stored in a variant type. Type $\langle \mathbf{q} : \langle \mathbf{m}_i : !S_i \times T_i \rangle_{i \in I} \rangle$ is read as: to send label \mathbf{m}_i to \mathbf{q} , (1) the channel vector should be “peeled off” from the nested record by extracting the field \mathbf{q} then \mathbf{m}_i ; then (2) it returns a pair $!S_i \times T_i$ of an output channel and a continuation. Type $\langle \mathbf{q} : ?[\mathbf{m}_i _ S_i \times T_i]_{i \in I} \rangle$ says that (1) the process extracts the value stored in the field \mathbf{q} , then reads on the resulting input channel (?) to receive a variant of type $[\mathbf{m}_i _ S_i \times T_i]_{i \in I}$; then, (2) the tag (constructor) \mathbf{m}_i of the received variant indicates the label which \mathbf{q} has sent, and the former’s argument S_i is the payload, and the latter T_i is the continuation.

The anti-symmetric structures between output types $\langle q:\langle m_i!S_i \times T_i \rangle_{i \in I} \rangle$ and input types $\langle q:?\langle m_i S_i \times T_i \rangle_{i \in I} \rangle$ (notice the placements of $!$ and $?$ symbol in these types) come from the fact that an output is an *internal choice* where output labels are proactively chosen via projection on a record field, while an input is an *external choice* where input labels are reactively chosen via pattern-matching among variant constructors.

3.2 Typing Global Combinators

A key finding of our work is that compatibility of local types can be checked using a type system with record and variant subtyping. Before explaining how each combinator ensures compatibility of types, we give an intuition of well-formed global protocols following [14].

Well-formedness and choice combinator. A well-formed global protocol ensures that a protocol can be correctly and *safely* realised by a system of endpoint processes. Moreover, a set of processes that follow the prescribed behaviour is *deadlock-free*. Well-formedness imposes several restrictions on the protocol structure, notably on *choices*. This is necessary because some protocols, such as `oAuth4` in Fig. 6(b) (§ 2), are unsafe or inconsistent. More precisely, a protocol is well-formed if local types can be generated for all of its roles, i.e the *endpoint projection* function [14, Def. 3.1][25] is defined for all roles. Our encoding allows the well-formedness restrictions to be checked *statically*, by the OCaml typechecker. Below, we explain the main syntactic restrictions of endpoint projection, which are imposed on *choices* and checked statically:

- R1 (active role)** in each branch of a choice, the first interaction is from the same sender role (*active role*) to the same receiver role (*directed output*).
- R2 (deterministic choice)** output labels from an active role are pairwise distinct (i.e., protocols are deterministic)
- R3 (mergeable)** the behaviour of a role from all branches should be mergeable, which is ensured by the following restrictions:
 - M1** two input choices are merged only if (1) their sender roles are the same (*directed input*), and (2) their continuations are recursively mergeable if labels are the same.
 - M2** two output choices can be merged only if they are the same.

Intuitively, the conditions in **R3** ensure that a process is able to determine unambiguously which branch of the choice has been taken by the active role, otherwise the process should be *choice-agnostic*, i.e it should perform the same actions in all branches. Requirement **R3** is known in the MPST literature as *recursive full merging* [14].

Typing system for global combinators. Deriving channel vector types from a global combinator corresponds to the *end point projection* in multiparty session types [21]. Projection of global protocols relies on the notion of merging (**R3**). As a result of the encoding of local types as channel vectors with record and variants, the *merging* relation coincides with the *least upper bound* (join) in the subtyping relation. This key observation allows us to embed well-formed global protocols in OCaml, and check them using the OCaml type system.

Next we give the typing system of global combinators, explaining how each of the typing rules ensures the verification conditions **R1-R3**. The typing system uses the following subtyping rules.

► **Definition 3.3.** The subtyping relation \sqsubseteq is coinductively defined by the following rules.

$$\frac{[\text{OSUB-}\bullet]}{\bullet \leq \bullet} \quad \frac{[\text{OSUB-OUTCH}]}{\#T \leq !T} \quad \frac{[\text{OSUB-OUT}]}{!T \leq !S} \quad \frac{[\text{OSUB-RCDDEPTH}]}{\langle \mathbf{1}_i : S_i \rangle_{i \in I} \leq \langle \mathbf{1}_i : T_i \rangle_{i \in I}} \quad \frac{[\text{OSUB-VAR}]}{[S_i \leq T_i]_{i \in I} \leq [S_i \leq T_i]_{i \in I}} \\ \frac{[\text{OSUB-INPCH}]}{\#T \leq ?T} \quad \frac{[\text{OSUB-INP}]}{?S \leq ?T} \quad \frac{[\text{OSUB-TUP}]}{S_1 \times \dots \times S_n \leq T_1 \times \dots \times T_n} \quad \frac{[\text{OSUB-}\mu\text{L}]}{\mu t.S \leq T} \quad \frac{[\text{OSUB-}\mu\text{R}]}{S \leq \mu t.T}$$

Among those, the rules $[\text{OSUB-}\mu\text{L}]$ and $[\text{OSUB-}\mu\text{R}]$ realise equi-recursive view of types. The only non-standard rule is $[\text{OSUB-RCDDEPTH}]$ which does not allow fields to be removed in the super type. This simulates OCaml's lack of row polymorphism where positive occurrences of objects are not allowed to drop fields. Note that the negative occurrences of objects in OCaml, which we use in process implementations, for example, do have row polymorphism, which correspond to standard record subtyping: $\frac{S_i \leq T_i \quad i \in I}{\langle \mathbf{1}_i : S_i \rangle_{i \in I \cup J} \leq \langle \mathbf{1}_i : T_i \rangle_{i \in I}}$. We use standard record subtyping, when typing processes. Since it permits removal of fields, it precisely simulates session subtyping on outputs. Typing rules for processes are left to [25].

The typing rules for global combinators (Fig. 7) are defined by the typing judgement of the form $\Gamma \vdash_{\mathbb{R}} \mathbf{g} : T$ where Γ is a type environment for recursion variables (definition follows), $\mathbb{R} = \mathbf{p}_1, \dots, \mathbf{p}_n$ is the sequence of roles which participate in \mathbf{g} , and $T = T_1 \times \dots \times T_n$ is a product of channel vector types where each T_i indicates a protocol which the role \mathbf{p}_i must obey. We use the product-based encoding to closely model our implementation and to avoid fixing the number of roles n of `finish` combinator by using *variable-length tuples* (see [25]).

► **Definition 3.4** (Global combinator typing rules). A *typing context* Γ is defined by the following grammar: $\Gamma ::= \emptyset \mid \Gamma, x : T$. The judgement $\Gamma \vdash_{\mathbb{R}} \mathbf{g} : T$ is defined by the rules in Fig. 7. We say \mathbf{g} is *typable with* \mathbb{R} if $\Gamma \vdash_{\mathbb{R}} \mathbf{g} : T$ for some Γ and T . If Γ is empty, we write $\vdash_{\mathbb{R}} \mathbf{g} : T$.

The rule $[\text{OTG-COMM}]$ states that \mathbf{p}_i has an output type $\langle \mathbf{p}_j : \langle \mathbf{m} : S \times T_i \rangle \rangle$ to \mathbf{p}_j with label \mathbf{m} , a payload typed by S and continuation typed by T_i ; a dual input type $\langle \mathbf{p}_i : ?[\mathbf{m} : S \times T_j] \rangle$ from \mathbf{p}_j and continuation typed by T_j ; and the rest of the roles are unchanged.

Rule $[\text{OTG-SUB}]$ is the key to obtain full merging using the subtyping relation, and along with the rule $[\text{OTG-CHOICE}]$, is a key to ensure the protocol is realisable, and free of communication errors. The rule $[\text{OTG-CHOICE}]$ requires (1) role \mathbf{p}_a to have an output type to the same destination role \mathbf{q} , which satisfies **R1**. The output labels $\{\mathbf{m}_k\}_{k \in K_i}$ are mutually disjoint at each branch \mathbf{g}_i , and are merged into a single record, which ensures that the choice is deterministic (**R2**). All other types stay the same, up to subtyping. Following requirement **M1** of **R3**, a non-directed external choices are prohibited. This is ensured by encoding the sender role of an input type as a record field, As the two different destination role labels would result in two record types with no join, following subtyping rule $[\text{OSUB-RCDDEPTH}]$, a non-directed external choices are safely reported as a type error. Non-directed internal choices are similarly prohibited (**M2**). On the other hand, directed external choices are allowed, as stipulated by **M1**, and ensured by the subtyping relation on variant types $[\text{OSUB-VAR}]$. For example, the two input types $\langle \mathbf{q} : ?[\mathbf{m}_1 : S_1 \times T_1] \rangle$ and $\langle \mathbf{q} : ?[\mathbf{m}_2 : S_2 \times T_2] \rangle$ can be unified as $\langle \mathbf{q} : ?[\mathbf{m}_i : S_i \times T_i]_{i \in \{1,2\}} \rangle$.

The rest of the rules are standard. Rule $[\text{OTG-FIX}]$ is for recursion; it assigns the recursion variable x a sequence of distinct fresh type variables in the continuation which is later looked up by $[\text{OTG-}x]$. In `tfix(t, T)`, we assign a unit type if the role does not contribute to the recursion (i.e., $T = \mathbf{t}'$ for any \mathbf{t}'), or forms a recursive type $\mu \mathbf{t}.T$ otherwise.

► **Example 3.5** (Typing a global combinator). We show that the global combinator $\mathbf{g}_{\text{Auth}} = (\mathbf{c} \rightarrow \mathbf{s}) \text{auth} (\text{choices } \mathbf{s} \{ (\mathbf{s} \rightarrow \mathbf{c}) \text{ok finish}, (\mathbf{s} \rightarrow \mathbf{c}) \text{cancel finish} \})$ has the following type under \mathbf{s}, \mathbf{c} :

$$\begin{array}{c}
 \text{[OTG-COMM]} \quad \Gamma \vdash_{\mathbb{R}} \mathbf{g} : (T_1 \times \dots \times T_i \times \dots \times T_j \times \dots \times T_n) \quad \mathbf{p}_i, \mathbf{p}_j \in \mathbb{R} \\
 \hline
 \Gamma \vdash_{\mathbb{R}} (\mathbf{p}_i \rightarrow \mathbf{p}_j) \mathbf{m} : S \mathbf{g} : (T_1 \times \dots \times \langle \mathbf{p}_j : \langle \mathbf{m} : !S \times T_i \rangle \rangle \times \dots \times \langle \mathbf{p}_i : \langle \mathbf{m} : !S \times T_j \rangle \rangle \times \dots \times T_n) \\
 \\
 \Gamma \vdash_{\mathbb{R}} \mathbf{g}_i : T_1 \times \dots \times T_{a-1} \times \langle \mathbf{q} : \langle \mathbf{m}_k : !S_k \times T'_k \rangle_{k \in K_i} \rangle \times T_{a+1} \times \dots \times T_n \\
 \text{[OTG-CHOICE]} \quad K_j \cap K_{j'} = \emptyset \text{ for all } j \neq j' \quad \forall i \in I \quad \mathbf{p}_a \in \mathbb{R} \\
 \hline
 \Gamma \vdash_{\mathbb{R}} \text{choice } \mathbf{p}_a \{ \mathbf{g}_i \}_{i \in I} : (T_1 \times \dots \times T_{a-1} \times \langle \mathbf{q} : \langle \mathbf{m}_k : !S_k \times T'_k \rangle_{k \in \bigcup_{i \in I} K_i} \rangle \times T_{a+1} \times \dots \times T_n) \quad \text{[OTG-x]} \\
 \hline
 \text{[OTG-finish]} \quad \text{[OTG-SUB]} \quad \Gamma \vdash_{\mathbb{R}} \mathbf{g} : S \leq T \quad \text{[OTG-fix]} \quad \Gamma, x : \mathbf{t}_{x1} \times \dots \times \mathbf{t}_{xn} \vdash_{\mathbb{R}} \mathbf{g} : T_1 \times \dots \times T_n \\
 \hline
 \Gamma \vdash_{\mathbb{R}} \text{finish} : \bullet \times \dots \times \bullet \quad \Gamma \vdash_{\mathbb{R}} \mathbf{g} : T \quad \Gamma \vdash_{\mathbb{R}} \text{fix } x \rightarrow \mathbf{g} : \text{tfix}(\mathbf{t}_{x1}, T_1) \times \dots \times \text{tfix}(\mathbf{t}_{xn}, T_n) \\
 \text{where } \mathbb{R} = \mathbf{p}_1, \dots, \mathbf{p}_n \text{ and, } \text{tfix}(\mathbf{t}, \mathbf{t}') = \bullet \text{ and } \text{tfix}(\mathbf{t}, T) = \mu \mathbf{t}. T \text{ otherwise.}
 \end{array}$$

■ **Figure 7** The typing rules for global combinators $\boxed{\Gamma \vdash_{\mathbb{R}} \mathbf{g} : T}$.

$\langle \mathbf{c} : \langle \text{auth}_T \times \langle \mathbf{c} : \langle \text{ok} : !T \times \bullet, \text{cancel} : !T \times \bullet \rangle \rangle \rangle \times \langle \mathbf{c} : \langle \text{auth} : !T \times \langle \mathbf{s} : \langle \text{ok}_T \times \bullet, \text{cancel}_T \times \bullet \rangle \rangle \rangle \rangle$

First, see that $\mathbf{g}_1 = ((\mathbf{s} \rightarrow \mathbf{c}) \text{ ok finish})$ has a typing derivation as follows (note that we omit the payload type T in global combinators):

$$\begin{array}{c}
 \vdash_{\mathbf{s}, \mathbf{c}} \text{finish} : \bullet \times \bullet \\
 \hline
 \vdash_{\mathbf{s}, \mathbf{c}} (\mathbf{s} \rightarrow \mathbf{c}) \text{ ok finish} : \langle \mathbf{c} : \langle \text{ok} : !T \times \bullet \rangle \rangle \times \langle \mathbf{s} : \langle \text{ok}_T \times \bullet \rangle \rangle
 \end{array}$$

For $\mathbf{g}_2 = ((\mathbf{s} \rightarrow \mathbf{c}) \text{ cancel finish})$ we have similar derivation. Then, type of role \mathbf{c} (the second of the tuple) is adjusted by [OTG-SUB], $\langle \mathbf{s} : \langle \text{ok}_T \times \bullet \rangle \rangle \leq \langle \mathbf{s} : \langle \text{ok}_T \times \bullet, \text{cancel}_T \times \bullet \rangle \rangle$ and $\langle \mathbf{s} : \langle \text{cancel}_T \times \bullet \rangle \rangle \leq \langle \mathbf{s} : \langle \text{ok}_T \times \bullet, \text{cancel}_T \times \bullet \rangle \rangle$, thus we have:

$$\begin{array}{c}
 \vdash_{\mathbf{s}, \mathbf{c}} \mathbf{g}_1 : \langle \mathbf{c} : \langle \text{ok} : !T \times \bullet \rangle \rangle \times \langle \mathbf{s} : \langle \text{ok}_T \times \bullet, \text{cancel}_T \times \bullet \rangle \rangle \\
 \vdash_{\mathbf{s}, \mathbf{c}} \mathbf{g}_2 : \langle \mathbf{c} : \langle \text{cancel} : !T \times \bullet \rangle \rangle \times \langle \mathbf{s} : \langle \text{ok}_T \times \bullet, \text{cancel}_T \times \bullet \rangle \rangle
 \end{array}$$

Then, by [OTG-CHOICE], we have the following derivation:

$$\begin{array}{c}
 \vdash_{\mathbf{s}, \mathbf{c}} \mathbf{g}_1 : \langle \mathbf{c} : \langle \text{ok} : !T \times \bullet \rangle \rangle \times \left\langle \mathbf{s} : \left[\begin{array}{c} \text{ok}_T \times \bullet, \\ \text{cancel}_T \times \bullet \end{array} \right] \right\rangle \quad \vdash_{\mathbf{s}, \mathbf{c}} \mathbf{g}_2 : \langle \mathbf{c} : \langle \text{cancel} : !T \times \bullet \rangle \rangle \times \left\langle \mathbf{s} : \left[\begin{array}{c} \text{ok}_T \times \bullet, \\ \text{cancel}_T \times \bullet \end{array} \right] \right\rangle \\
 \hline
 \vdash_{\mathbf{s}, \mathbf{c}} \text{choices } \{ \mathbf{g}_1, \mathbf{g}_2 \} : \langle \mathbf{c} : \langle \text{ok} : !T \times \bullet, \text{cancel} : !T \times \bullet \rangle \rangle \times \langle \mathbf{s} : \langle \text{ok}_T \times \bullet, \text{cancel}_T \times \bullet \rangle \rangle
 \end{array}$$

Note that, in the above premises, the first element of the tuple specifying the behaviour of choosing role \mathbf{s} , namely $\langle \mathbf{c} : \langle \text{ok} : !T \times \bullet \rangle \rangle$ and $\langle \mathbf{c} : \langle \text{cancel} : !T \times \bullet \rangle \rangle$, are disjointly combined into $\langle \mathbf{c} : \langle \text{ok} : !T \times \bullet, \text{cancel} : !T \times \bullet \rangle \rangle$ in the conclusion. Then, by applying [OTG-COMM] again, we get the type for \mathbf{g}_{Auth} presented above.

3.3 Evaluating Global Combinators to Channel Vectors

Channel vectors are data structures which are created from a global combinator at initialisation, and used for sending/receiving values from/to participants. Channel vectors implement multiparty communications as nested binary io-typed channels.

► **Definition 3.6** (Channel vectors). *Channel vectors* (c, c', \dots) and *wrappers* (h, h', \dots) are defined as:

$$\begin{array}{l}
 c, c' ::= v, \dots \mid s, s', \dots \mid (c_1, \dots, c_n) \mid [1=c] \mid \langle \mathbf{1}_i = c_i \rangle_{i \in I} \mid \mu x. c \mid [s_i @ h_i]_{i \in I} \\
 h, h' ::= [] \mid [1=h] \mid (c_1, \dots, h_k, \dots, c_n) \mid \langle \mathbf{1}_1 = c_1, \dots, \mathbf{1}_k = h, \dots, \mathbf{1}_n = c_n \rangle \quad \mathbf{1} ::= \mathbf{p} \mid \mathbf{m}
 \end{array}$$

Channel vectors c are either *base values* v or *runtime values* generated from global combinators which include *names* (simply-typed binary channels) s, s', \dots , *tuples* (c_1, \dots, c_n) , *variants* $[1=c]$, *records* $\langle \mathbf{1}_i = c_i \rangle_{i \in I}$, and *recursive values* $\mu x. c$ where x is a bound variable.

We introduce an extra runtime value, *wrapped names* $[s_i @ h_i]_{i \in I}$, inspired by Concurrent ML's `wrap` and `choose` functions [45], which are a sequence $[...]_{i \in I}$ of pairs of input name s_i and a wrapper h_i . A wrapper h contains a single hole $[]$. An input on wrapped names

$$\begin{array}{c}
\frac{[\text{OTC-S}]}{\Gamma, s : \#T \vdash s : \#T} \quad \frac{[\text{OTC-X}]}{\Gamma, x : T \vdash x : T} \quad \frac{[\text{OTC-0}]}{\Gamma \vdash () : \bullet} \quad \frac{[\text{OTC-SUB}]}{\Gamma \vdash c : T} \quad \frac{[\text{OTC-TUP}]}{\Gamma \vdash (c_1, \dots, c_n) : T_1 \times \dots \times T_n} \\
\frac{[\text{OTC-VARIANT}]}{\Gamma \vdash [l=c] : [l_T]} \quad \frac{[\text{OTC-RECORD}]}{\Gamma \vdash \langle l_i = c_i \rangle_{i \in I} : \langle l_i : T_i \rangle_{i \in I}} \quad \frac{[\text{OTC-}\mu]}{\Gamma \vdash \mu x. c : \mu t. T} \\
\frac{[\text{OTC-WRAPINP}]}{\Gamma \vdash s_i : ?S_i \quad \Gamma \vdash h_i : T[S_i] \quad \forall i \in I} \quad \frac{[\text{OTC-WRAPPER}]}{\Gamma \vdash h : T[T']}
\end{array}$$

■ **Figure 8** The typing rules for channel vectors and wrappers $\boxed{\Gamma \vdash c : T} \quad \boxed{\Gamma \vdash h : H}$.

$[s_i @ h_i]_{i \in I}$ is *multiplexed* over the set of names $\{s_i\}_{i \in I}$. When a sender outputs value c' on name s_j ($j \in I$), the corresponding input waiting on $[s_i @ h_i]_{i \in I}$ yields a value $h_j[c']$ where the construct $h[c]$ denotes a value obtained by replacing the hole $[]$ in h with c (i.e. applying function h to c). We write $[l_i = (s_i, c_i)]_{i \in I}$ for $[s_i @ [l_i = ([], c_i)]]_{i \in I}$.

► **Definition 3.7** (Typing rules for channel vectors). Fig. 8 gives the typing rules for channel vectors and wrappers. The typing judgement for (1) channel vectors has the form $\Gamma \vdash c : T$; (2) wrappers has the form $\Gamma \vdash h : H$ where the type for wrappers is defined as $H ::= T[S]$; We assume that all types in Γ are closed.

The rules for channel vectors are standard where the subtyping relation in rule $[\text{OTC-SUB}]$ is defined at Definition 3.3. For wrappers, rule $[\text{OTC-WRAPINP}]$ types wrapped names where the payload type S' of input channel s is the same as the hole's type, and all wrappers have the same result type T . Rule $[\text{OTC-WRAPPER}]$ checks type of a channel vector $c = h[x]$ and replaces x with the hole $[]$.

Evaluation of global combinators is the key to implement a multiparty protocol to a series of binary, simply-typed communications based on channel vectors. We define $\llbracket \mathbf{g} \rrbracket_{\mathbb{R}}^s$ where \mathbb{R} is a sequence of roles in \mathbf{g} and s is a *base name* freshly assigned to an initiation expression at runtime. The generated channels are interconnected to each other and the created channel vectors are distributed and shared among expressions running in parallel, enabling them to interact via binary names.

The followings are basic operations on records, tuples and recursive values which are used to define evaluations of global combinators.

► **Definition 3.8** (Operations). (1) The *unfolding* $\text{unfold}^*(c)$ of a recursive value is defined by the smallest n such that $\text{unfold}^n(c) = \text{unfold}^{n+1}(c)$, and $\text{unfold}(\cdot)$ is defined as:

$$\text{unfold}(\mu x. c) = c\{\mu x. c/x\} \quad \text{unfold}(c) = c \quad \text{otherwise}$$

where $f^{n+1}(x) = f(f^n(x))$ for $n \geq 2$ and $f^1(x) = f(x)$. (2) $c \# \mathbf{1}$ denotes the **record projection**, which projects on field $\mathbf{1}$ of record value c , defined as: $\langle l_i = c_i \rangle_{i \in I} \# \mathbf{1}_k = \text{unfold}^*(c_k)$, where $\#$ is left-associative, i.e. $c \# \mathbf{1}_1 \# \dots \# \mathbf{1}_n = ((\dots(c \# \mathbf{1}_1) \# \dots) \# \mathbf{1}_n)$. (3) The i -th projection on a tuple, $c(i)$ is defined as $(c_1, \dots, c_n)(i) = c_i$ for $1 \leq i \leq n$. (4) $\text{fix}(x, x') = ()$; otherwise $\text{fix}(x, c) = \mu x. c$.

► **Definition 3.9** (Evaluation of a global combinator). Given \mathbb{R} and fresh s , the *evaluation* $\llbracket \mathbf{g} \rrbracket_{\mathbb{R}}^s$ of global combinator \mathbf{g} is defined in Fig. 9. We write $\llbracket \mathbf{g} \rrbracket^s$ if $\mathbb{R} = \text{roles}(\mathbf{g})$.

The evaluation for communication $(\mathbf{p}_j \rightarrow \mathbf{p}_k) \mathbf{m} : S \mathbf{g}$ connects between \mathbf{p}_j and \mathbf{p}_k by the name $s_{\{\mathbf{p}_j, \mathbf{p}_k, \mathbf{m}, i\}}$ by wrapping j -th and k -th channel vector with an output and an input structure, respectively. The name $s_{\{\mathbf{p}_j, \mathbf{p}_k, \mathbf{m}, i\}}$ is indexed by two role names $\mathbf{p}_j, \mathbf{p}_k$, label \mathbf{m} and an index i so that (1) it is only shared between two roles \mathbf{p}_j and \mathbf{p}_k , (2) communication only occurs when it tries to communicate a specific label \mathbf{m} , and (3) both the sender and

$$\begin{aligned}
 \llbracket (\mathbf{p}_j \rightarrow \mathbf{p}_k) \mathbf{m} : S \mathbf{g} \rrbracket_{\mathbb{R}}^s &= \\
 &\left(\llbracket \mathbf{g} \rrbracket_{\mathbb{R}}^s(1), \dots, \llbracket \mathbf{g} \rrbracket_{\mathbb{R}}^s(j-1), \left\langle \mathbf{p}_k = \left\langle \mathbf{m} = (s_{\{\mathbf{p}_j, \mathbf{p}_k, \mathbf{m}, i\}}, \llbracket \mathbf{g} \rrbracket_{\mathbb{R}}^s(j)) \right\rangle \right\rangle, \llbracket \mathbf{g} \rrbracket_{\mathbb{R}}^s(j+1), \right. \\
 &\quad \left. \dots, \llbracket \mathbf{g} \rrbracket_{\mathbb{R}}^s(k-1), \left\langle \mathbf{p}_j = \left[\mathbf{m} = (s_{\{\mathbf{p}_j, \mathbf{p}_k, \mathbf{m}, i\}}, \llbracket \mathbf{g} \rrbracket_{\mathbb{R}}^s(k)) \right] \right\rangle, \llbracket \mathbf{g} \rrbracket_{\mathbb{R}}^s(k+1), \dots, \llbracket \mathbf{g} \rrbracket_{\mathbb{R}}^s(n) \right) \\
 &\quad \text{where } i \text{ is fresh.} \\
 \llbracket \text{choice } \mathbf{p}_a \{ \mathbf{g}_i \}_{i \in I} \rrbracket_{\mathbb{R}}^s &= \\
 &\left(\bigsqcup_{i \in I} (\llbracket \mathbf{g}_i \rrbracket_{\mathbb{R}}^s(1)), \dots, \bigsqcup_{i \in I} (\llbracket \mathbf{g}_i \rrbracket_{\mathbb{R}}^s(a-1)), \left\langle \mathbf{q} = \left\langle \mathbf{m}_k = c_k \right\rangle_{k \in K} \right\rangle, \bigsqcup_{i \in I} (\llbracket \mathbf{g}_i \rrbracket_{\mathbb{R}}^s(a+1)), \dots, \bigsqcup_{i \in I} (\llbracket \mathbf{g}_i \rrbracket_{\mathbb{R}}^s(n)) \right) \\
 &\quad \text{where } \text{unfold}^*(\llbracket \mathbf{g}_i \rrbracket_{\mathbb{R}}^s(a)) = \left\langle \mathbf{q} = \left\langle \mathbf{m}_k = c_k \right\rangle_{k \in K_i} \right\rangle \text{ and } K = \bigcup_{i \in I} K_i \\
 \llbracket \text{fix } x \rightarrow \mathbf{g} \rrbracket_{\mathbb{R}}^s &= (\text{fix}(x_1, \llbracket \mathbf{g} \rrbracket_{\mathbb{R}}^s(1)), \dots, \text{fix}(x_n, \llbracket \mathbf{g} \rrbracket_{\mathbb{R}}^s(n))) \\
 \llbracket x \rrbracket_{\mathbb{R}}^s &= (x_1, \dots, x_n) \quad \llbracket \text{finish} \rrbracket_{\mathbb{R}}^s = (\text{()}, \dots, \text{()})
 \end{aligned}$$

■ **Figure 9** Evaluation of global combinators $\llbracket \mathbf{g} \rrbracket_{\mathbb{R}}^s$.

the receiver agree on the payload type. Here, the index i is used to distinguish between names generated from the same label \mathbf{m}' but different payload type $\mathbf{m} : T$ and $\mathbf{m} : T'$, ensuring consistent typing of generated channel vectors. The choice combinator $\text{choice } \mathbf{p}_a \{ \mathbf{g}_i \}_{i \in I}$ extracts the output channel vector (i.e. the nested records of the form $\langle \mathbf{q} = \langle \mathbf{m}_k = c_k \rangle_{k \in K_i} \rangle$) at \mathbf{p}_a from each branch \mathbf{g}_i , and merges them into a single output. Channel vectors for the other roles are merged by $c_1 \sqcup c_2$ where merging for the outputs is an intersection of branchings from c_1 and c_2 , while merging of the inputs is their union. We explain merging by example (Example 3.10) and leave the full definition in [25].

For the recursion combinator, function $\text{fix}(x_i, c_i)$ forms a recursive value for repetitive session, or voids it as $()$ if it does not contain any names.

► **Example 3.10** (Global combinator evaluation). Let $s_1 = s_{\{\mathbf{c}, \mathbf{s}, \text{ok}, 0\}}$, $s_2 = s_{\{\mathbf{c}, \mathbf{s}, \text{cancel}, 0\}}$ and $s_3 = s_{\{\mathbf{s}, \mathbf{c}, \text{auth}, 0\}}$. Then:

$$\begin{aligned}
 &\llbracket \mathbf{g}_{\text{Auth}} \rrbracket_{\mathbb{R}}^s \\
 &= \llbracket (\mathbf{c} \rightarrow \mathbf{s}) \text{auth} (\text{choices } \mathbf{s} \{ (\mathbf{s} \rightarrow \mathbf{c}) \text{ok finish}, (\mathbf{s} \rightarrow \mathbf{c}) \text{cancel finish} \}) \rrbracket_{\mathbb{R}}^s \\
 &= \left(\text{Here, we have } \left\{ \begin{array}{l} \mathbf{g}_L = (\mathbf{c} \rightarrow \mathbf{s}) \text{ok finish}, \quad \mathbf{g}_R = (\mathbf{c} \rightarrow \mathbf{s}) \text{cancel finish}, \\ \llbracket \mathbf{g}_L \rrbracket_{\mathbb{R}}^s = \langle \langle \mathbf{s} = [\text{ok} = (s_1, \text{()})] \rangle, \langle \mathbf{c} = \langle \text{ok} = (s_1, \text{()}) \rangle \rangle \rangle, \\ \llbracket \mathbf{g}_R \rrbracket_{\mathbb{R}}^s = \langle \langle \mathbf{s} = [\text{cancel} = (s_2, \text{()})] \rangle, \langle \mathbf{c} = \langle \text{cancel} = (s_2, \text{()}) \rangle \rangle \rangle, \end{array} \right. \right), \\
 &\quad \left. \text{concatenating } \left\{ \begin{array}{l} \text{unfold}^*(\llbracket \mathbf{g}_L \rrbracket_{\mathbb{R}}^s(2)) = \llbracket \mathbf{g}_L \rrbracket_{\mathbb{R}}^s(2) = \langle \mathbf{s} = \langle \text{ok} = c_{L2} \rangle, c_{L2} = (s_1, \text{()}) \rangle, \\ \text{unfold}^*(\llbracket \mathbf{g}_R \rrbracket_{\mathbb{R}}^s(2)) = \llbracket \mathbf{g}_R \rrbracket_{\mathbb{R}}^s(2) = \langle \mathbf{s} = \langle \text{cancel} = c_{R2} \rangle, c_{R2} = (s_2, \text{()}) \rangle \end{array} \right\} \right) \\
 &= \langle \langle \mathbf{s} = \langle \text{auth} = (s_3, \llbracket \mathbf{g}_L \rrbracket_{\mathbb{R}}^s(1) \sqcup \llbracket \mathbf{g}_R \rrbracket_{\mathbb{R}}^s(1)) \rangle, \langle \mathbf{c} = [\text{auth} = (s_3, \langle \mathbf{c} = \langle \text{ok} = c_{L2}, \text{cancel} = c_{R2} \rangle \rangle) \rangle] \rangle \rangle \\
 &= \left(\langle \langle \mathbf{s} = \langle \text{auth} = (s_3, \langle \mathbf{s} = [\text{ok} = (s_1, \text{()})], \text{cancel} = (s_2, \text{()}) \rangle \rangle \rangle \rangle, \right. \\
 &\quad \left. \langle \mathbf{c} = [\text{auth} = (s_3, \langle \mathbf{c} = \langle \text{ok} = (s_1, \text{()}) \rangle, \text{cancel} = (s_2, \text{()}) \rangle \rangle) \rangle] \rangle \right)
 \end{aligned}$$

The following main theorem states that if a global combinator is typable, the generated channel vectors are well-typed under the corresponding local types.

► **Theorem 3.11** (Realisability of global combinators). If $\vdash_{\mathbb{R}} \mathbf{g} : T$, then $\llbracket \mathbf{g} \rrbracket_{\mathbb{R}}^s = c$ is defined and $\{s_i : S_i\}_{s_i \in \text{fn}(c)} \vdash c : T$ for some $\{\tilde{S}_i\}$.

This property offers the type soundness and communication safety for `ocaml-mpst` endpoint programs: a statically well-typed `ocaml-mpst` program will satisfy subject reduction theorem and never performs a non-compliant I/O action w.r.t. the underlying binary channels. We leave the formal definition of `ocaml-mpst` endpoint programs, operational semantics, typing system, and the subject reduction theorem in [25].

Global Combinator	Type
<code>finish</code>	$(\text{close} * \dots * \text{close})$
$(r_i \text{ --> } r_j) \text{ m } g$	Given $g : (t_{r_1} * \dots * t_{r_n})$, Return $(t_{r_1} * \dots * \langle r_j : \langle \text{m} : ('v * t_{r_i}) \text{ out} \rangle * \dots * \langle r_i : [\text{>} \text{m of } 'v * t_{r_j}] \text{ inp} \rangle * \dots * t_{r_n})$
<code>choice_at</code> $r_a \text{ mrg}$ (r_a, g_1) (r_a, g_2)	Given $1 \leq a \leq n$, $g_1 : (t_{r_1} * \dots * t_{r_{a-1}} * \langle r_b : \langle \text{m}_i : (v_i, s_i) \text{ out} \rangle_{i \in I} \rangle * t_{r_{a+1}} * \dots * t_{r_n})$, $g_2 : (t_{r_1} * \dots * t_{r_{a-1}} * \langle r_b : \langle \text{m}_j : (v_j, s_j) \text{ out} \rangle_{j \in J} \rangle * t_{r_{a+1}} * \dots * t_{r_n})$, and $\text{mrg} : \text{a concatenator ensuring the two label sets are mutually disjoint } (I \cap J = \emptyset)$, Return $(t_{r_1} * \dots * t_{r_{a-1}} * \langle r_b : \langle \text{m}_k : (v_k, s_k) \text{ out} \rangle_{k \in I \cup J} \rangle * t_{r_{a+1}} * \dots * t_{r_n})$
<code>fix</code> $(\text{fun } x \text{ -> } g)$	Given $g : (t_{r_1} * \dots * t_{r_n})$ under assumption that $x : (t_{r_1} * \dots * t_{r_n})$, x is guarded in g Return $(t_{r_1} * \dots * t_{r_n})$
<code>closed_at</code> $r_a \text{ } g$	Given $g : (t_{r_1} * \dots * t_{r_{a-1}} * \text{close} * t_{r_{a+1}} * \dots * t_{r_n})$ and $1 \leq a \leq n$, Return $(t_{r_1} * \dots * t_{r_{a-1}} * \text{close} * t_{r_{a+1}} * \dots * t_{r_n})$

■ **Figure 10** Type of Global Combinators in OCaml.

4 Implementing Global Combinators

We give a brief overview on the type manipulation techniques that enable type checking of global combinators in native OCaml. § 4.1 gives a high-level intuition of our approach, § 4.2 illustrates evaluation of global combinators to channel vectors in pseudo OCaml code, and § 4.3 presents the typing of global combinators in OCaml. Furthermore, in [25], we develop *variable-length tuples* using state-of-the-art functional programming techniques, e.g., GADT and polymorphic variants, to improve usability of `ocaml-mpst`.

4.1 Typing Global Combinators in OCaml: A Summary

In Fig. 10 we illustrate the type signature of each global combinator, which is a transliteration of the typing rules (Fig. 7) into OCaml. In the figure, OCaml type $(t_{r_1} * \dots * t_{r_n})$ corresponds to a n -tuple of channel vector types $t_{r_1} \times \dots \times t_{r_n}$. The implementation makes use of *variable-length tuples* to represent tuples of channel vectors, and therefore the developer does not have to explicitly specify the number of roles n (see [25]). A few type-manipulation techniques are expanded later in § 4.3. Henceforth, we only make a few remarks, regarding some discrepancies with the implementation.

OCaml types	Types in § 3
$\langle \text{r} : [\text{>} \text{m}_i \text{ of } v_i * t_i]_{i \in I} \text{ inp} \rangle$	$\langle \text{r} : ?[\text{m}_i _ S_i \times T_i]_{i \in I} \rangle$
$\langle \text{r} : \langle \text{m}_i : (v_i, t_i) \text{ out} \rangle_{i \in I} \rangle$	$\langle \text{r} : \langle \text{m}_i : !S_i \times T_i \rangle_{i \in I} \rangle$
<code>close</code> (=unit)	•
<code>t as 'x</code>	$\mu x. T$

Channel vector types in OCaml. The OCaml syntax of channel vector types is given on the right. The difference with its formal counterparts are minimal. In particular, records are implemented using OCaml object types, and record fields correspond to object methods, i.e. `role_q` is a method. In type $[\text{>} \text{m}_i \text{ of } t_i]_{i \in I}$, the symbol > marks an *open* polymorphic variant type which can have more tags. The types `inp` and `out` stand for an input and output types with a payload type v_i and a continuation t_i . Recursive channel vector types are implemented using OCaml equi-recursive types.

On branching and compatibility checking. As we explained in § 3.2, branching is the key to ensure the protocol is realisable, and free of communication errors. To ensure that the choice is deterministic, it must be verified that the set of labels in each branch are disjoint. Since OCaml objects do not support *concatenation* (combining of multiple methods e.g., [57, 19]), and cannot automatically verify that the set of labels (encoded as object methods) are disjoint, the user has to manually write a disjoint merge function *mrg* that concatenates two objects with different methods into one (see [25] for examples). This part can be completely automated by PPX syntactic extension in OCaml. On compatibility checking of non-choosing roles, external choice $\langle r: [>^m1 \text{ of } \dots] \text{ inp} \rangle$ and $\langle r: [>^m2 \text{ of } \dots] \text{ inp} \rangle$, the types can be recursively merged by OCaml type inference to $\langle r: [>^m1 \text{ of } \dots |^m2 \text{ of } \dots] \text{ inp} \rangle$ thanks to the row polymorphism on polymorphic variant types ($>$), while non-directed external choices and other incompatible combination of types (e.g., input and output, input and closing, and output and closing) are statically excluded.

On unguarded recursion. The encoding of recursion `fix (fun x -> g)` has two caveats w.r.t the typing system: (1) OCaml does not check if a recursion is guarded, thus for example `fix (fun x -> x)` is allowed. We cannot use OCaml value recursion, because global combinators generate channels at run-time. (2) Even if a loop is guarded, Hindley-Milner type inference may introduce arbitrary local type at some roles. For example, consider the global protocol `fix (fun x -> (ra --> rb) msg x)` which specifies an infinite loop for roles $\notin \{r_a, r_b\}$, and does not specify any behaviour for any other roles. To prevent undefined behaviour, the typing rule marks the types of the roles that are not used as closed `tfix(t, T)`. Unfortunately, in type inference, we do not have such control, and the above protocol will introduce a polymorphic type t_{r_i} for role $r_i \notin \{r_a, r_b\}$, which can be instantiated by *any* local type.

Fail-fast policy. We regard the above intricacies on recursion as a *fact of life* in any programming language, and provide a few workarounds. For (1), we adopt a “fail-fast” policy: Our library throws an exception if there is an unguarded occurrence of a recursion variable. This check is performed when evaluating a global combinator before any communication is started. As for (2), we require the programmer to adhere to a coding convention when specifying an infinite protocol. They have to insert additional combinator `closed_at ra g`, which consistently instantiates type variable t_{r_a} with `close`, leaving other roles intact. If the programmer forgets this insertion, fail-fast approach applies, and our library throws a runtime exception before the protocol has started. In addition, self-sent messages `(r -->r)msg` for any `r` are reported as an error at runtime.

4.2 Implementing Global Combinator Evaluation

Following § 3.3, in Fig. 11, we illustrate the implementation of the global combinators, by assuming that method names and variant tags are *first class* in this pseudo-OCaml. Communication combinator `(-->)` is presented in Fig. 11 (a) where the communication combinator `((ri --> rj) m g)` yields two reciprocal channel vectors of type $\langle r_j: \langle m: (v, t_{r_i}) \text{ out} \rangle >$ and $\langle r_i: [>^m \text{ of } v * t_{r_j}] \text{ inp} \rangle$.

The implementation starts by extracting the continuations (the channel vectors) at each role (Line 3). Line 4 creates a fresh new channel `s` of a polymorphic type `v channel` shared among two roles, which is a source of type safety regarding *payload* types. Line 6 creates an output channel vector. We use a shorthand $\langle m = e \rangle$ to represent an OCaml object method $m = e$ `end`. Thus, it is bound to c_{r_i} , by nesting the pair `(s, cri)` inside two

```

1 let (-->) ri rj m g =
2 (* extract the continuations *)
3 let (cr1, cr2, ... , crn) = g in
4 let s = Event.new_channel () in
5 (* create an output channel vector *)
6 let cri = (<rj = <m = (s, cri)>>) in
7 (* create an input channel vector *)
8 let crj = (<ri =
9   Event.wrap s (fun x -> `m(x, crj)) >) in
10 (cr1, cr2, ... , crn)

let choice_at ra mrg g1 g2 =
let (c1r1, c1r2, ... , c1rn) = g1 in
let (c2r1, c2r2, ... , c2rn) = g2 in
let cra =
  (concatenate c1ra and c2ra using mrg) in
let cr1 = merge c1r1 c2r1 in
let cr2 = merge c1r2 c2r2 in
(* .. repeatedly merge each ri ≠ ra .. *)
let crn = merge c1rn c2rn in
(cr1, cr2, ... , crn)

```

■ **Figure 11** Implementation of communication combinator and (a) branching combinator (b).

objects, one with a method role, and another with a method label, forming type $\langle r_j : \langle m : (\nu, t_{r_i}) \text{out} \rangle \rangle$. Similarly, Line 8 creates an input channel vector c_{r_j} , by wrapping channel s in a polymorphic variant using `Event.wrap` from Concurrent ML and nesting it in an object type, forming type $\langle r_i : [\nu^m \text{ of } \nu^* t_{r_j}] \text{inp} \rangle$. This wrapping relates tag m and continuation t_j to the input side, enabling external choice when merged. Finally, the newly updated tuple of channel vectors is returned (Line 10).

Fig. 11 (b) illustrates the choice combinator `choice_at`. Line 6–9 specifies that the channel vectors at non-choosing roles are *merged*, using a `merge` function. Intuitively, `merge` does a type-case analysis on the type of channel vectors, as follows: (1) for an input channel vector, it makes an *external choice* among (wrapped) input channels, using the `Event.choose` function from Concurrent ML; (2) for an output channel vector, the bare channel is *unified* label-wise, in the sense that an output on the unified channel can be observed on both input sides, which is achieved by having channel type around a reference cell; and (3) handling of channel vector of type `close` is trivial.

First-class methods. Method names r_i , r_j and m and the variant tag m occurring in $((r_i \text{-->} r_j) m g)$ are assumed in § 4.1 to be first-class values. Since such behaviour is not readily available in vanilla OCaml, we simulate it by introducing the type `method_` (Line 2 in Fig. 12), which creates values that behave like method objects. The type is a record with a *constructor function* `make_obj` and a *destructor function* `call_obj` (see example in Lines 3–6). We use that idea to implement labels and roles as object methods. The encoding of local

```

1 (* the definition of the type method_ *)
2 type ('obj, 'mt) method_ = {make_obj: 'mt -> 'obj; call_obj: 'obj -> 'mt}
3 (* example usage of method_: *)
4 val login_method : (<login : 'mt>, 'mt) method_ (* the type of login_method *)
5 let login_method =
6   {make_obj=(fun v -> object method login = v end); call_obj=(fun obj -> obj#login)}
7
8 (* the definition of the type label *)
9 type ('obj, 'ot, 'var, 'vt) label = {obj: ('obj, 'ot) method_; var: 'vt -> 'var}
10 (* example usage of label *)
11 val login : (<login : 'mt>, 'mt, [> `login of 'vt], 'vt) label
12 let login = {obj=login_method; var=(fun v -> `login(v))}
13
14 (* example usage of role: *)
15 let s = {index=Zero;
16   label={make_obj=(fun v -> object method role_S=v end); call_obj=(fun o -> o#role_S)}}

```

■ **Figure 12** Implementation of first-class methods and labels.

types stipulates that labels are object methods (in case of internal choice) and as variant tags (in case of external choice). Hence, the `label` type (Line 9 in Fig. 12), is defined as a pair of a first-class method, i.e using `method_`, and a *variant constructor function*. While object and variant constructor functions are needed to compose a channel vector in (`-->`), object destructor functions are used in `merge` in `choice_at`, to extract bare channels inside an object. Variant destructors are not needed, as they are destructed via pattern-matching and merging is done by `Event.choose` of Concurrent ML. Roles are defined similarly to labels. See example in Line 15 (the full definition of `role` type is available in [25]).

4.3 Typing Global Combinators via Polymorphic Lenses

This section shows one of our main implementation techniques – the use of *polymorphic lenses* [16, 42] for *index-based updates* on tuple types. This is essential to the implementation of the typing of Fig. 10 in OCaml. To demonstrate our technique, we sketch the type of the branching combinator, in a simplified form. The types of all combinators, incorporating first-class methods and variable-length tuples, can be found in [25]. The branching combinator demonstrates our key observation that merging of local types can be implemented using row polymorphism in OCaml, which simulates the least upper bound on channel vector types.

Intuitively, a lens is a functional pointer, often utilised to access and modify elements of a nested data structure. In our implementation, lenses provide a way to *update* a channel vector in a tuple $(t_{r_1} * \dots * t_{r_n})$. The type of the lens $(\text{'g0', 't0', 'g1', 't1}) \text{idx}$ itself points to an element in a specific position in a tuple, by denoting that “an element `'t0` is in a tuple `'g0`” in a type-parametric way. Furthermore, this polymorphic lens is capable to express updating the *type* of an element, from `'t0` in tuple `'g0` to `'t1`, which will update `'g0` itself to `'g1`. More precisely, the `idx` type has two operations:

`get: ('g0, 't0, _, _) idx -> 'g0 -> 't0` and `put: ('g0, _, 'g1, 't1) idx -> 'g0 -> 't1 -> 'g1`.

For example, a lens pointing to the first element of a 3-tuple has the type $((\text{'x*'a*'b}), \text{'x}, (\text{'y*'a*'b}), \text{'y}) \text{idx}$.

The branching combinator `choice_at` $r_a \text{ mrg } (r_a, g_1) (r_a, g_2)$ is declared in following way:

```

1 val choice_at : ('g0, close, 'g, 'tlr) idx -> (* the index of the selecting role *)
2   ('tlr, 't1, 'tr) disj -> (* the type of disjoint merge function *)
3   ('g1, 't1, 'g0, close) idx * 'g1 -> (* the type of the first tuple *)
4   ('gr, 'tr, 'g0, close) idx * 'gr -> (* the type of the second tuple *)
5   'g (* the type of the result tuple *)

```

The type variables in the above is resolved *a la* logic programs in Prolog, where several type variables are unified to compose a tuple type of channel vectors. It requires that both continuation tuples `'g1` and `'gr` should be of the same type, *except for* the position of active role r_a . The two `idx` types paired with continuations force this unification, by putting `close` at r_a in `'g1` and `'gr`. Thus, the result type `'g0` is shared among both lenses, so that it contains only types of non-choosing roles and `close`. Each element in `'g0` is then pairwise merged⁴. The result type of the combinator `'g` is obtained by modifying the merged tuple of channel vectors `'g0` by updating the type of the active role r_a from `close` to `'tlr`, which is the result type of the object concatenation function `mrg`. Function `mrg` takes the channel vector types for the role r_a in `g1` and `g2`, namely `'t1` and `'tr`, and returns the result type `'tlr`. The signature of the combinator also explains the extra occurrence roles paired with each branch. Since we need lens r_a within three *different instantiations* for different element types `'t1`, `'tr` and `'tlr` at the position r_a , we need three occurrences of the same lens.

⁴ We have implemented the type-case analysis for `merge` mentioned in § 4.2 via a wrapper called *mergeable* around each channel vector, which bundles a channel vector and its *merging strategy*.

Dynamic	Static
<code><role_q: <m: ('v, 't) out>></code>	<code><role_p: <m: ('v data, 't) out>> lin</code> (base value) <code><role_p: <m: ('s lin, 't) out>> lin</code> (delegation)
<code><role_p: [~m of 'v* 't] inp></code>	<code><role_p: [~m of 'v data* 't lin] inp lin> lin</code> (base value) <code><role_p: [~m of 's lin* 't lin] inp lin> lin</code> (delegation)
<code>close</code>	<code>close lin</code>

■ **Figure 13** Channel Vector Types with (a) Dynamic and (b) Static Linearity Checks.

5 Dynamic and Static Linearity Checks in the Communication API

To ensure that an implementation faithfully implements a well-formed, safe global protocol, MPST theory requires that all communication channels are used linearly. Similarly, the safety of our library depends on the linear usage of channels. Our library offers two mechanisms for checking that a channel is used linearly: *static* and *dynamic*. Here, we briefly explain each of these mechanisms, by comparing their API usages in Fig. 14 and types in Fig. 13, where the dynamic version stays on the left while the static one is on the right.

Dynamic Linearity Checking. Dynamic checking, where linearity violations are detected at runtime, is proposed by [55] and [22], and later adopted by [41, 47]. In `ocaml-mpst`, dynamic linearity checking is implemented by wrapping the input and output channels, with a boolean flag that is set to true once the channel has been used. If linearity is violated, i.e a channel is accessed after the linearity flag has been set to true, then an exception `InvalidEndpoint` will be raised. Note that our library correctly handles output channels between several alternatives being used *only once*; for example, from a channel vector c of type `<r: <ok: (string, close) out; cancel: (string, close) out>>`, the user can extract two channels `c#r#ok` and `c#r#cancel` where an output must take place on either of the two bare channels, but not both. In addition, our library wraps each bare channel with a *fresh* linearity flag on each method invocation, since in recursive protocols, a bare channel is often *reused*, as the formalism (§ 3) implies.

Static Linearity Checking with Monads and Lenses. The static checking is built on top of `linocaml` [24]: a library implementation of linear types in OCaml which combines the usage of *parameterised monads* [2] and polymorphic lenses (see § 4.3), to enable static type-checking on the linear usage of channels. In particular, we reuse several techniques from [24, 27]. A parameterised monad, which we model by the type $((pre, post, v) \text{ monad})$, denotes a computation of type v with a *pre*- and a *post*-condition, and they are utilised to track the creation and consumption of resources at the type level. A well-known restriction of parameterised monads in the context of session types, is that they support communication on a single channel only, and hence are incapable of expressing session delegation and/or interleaving of multiple session channels. To overcome this limitation, the *slot monad* proposed in [24, 27] extends the parameterised monad to denote *multiple* linear resources in the *pre*- and *post*-conditions. The resources are represented as a sequence, and each element is modified using polymorphic lenses [42].

We incorporate the above-mentioned techniques of `linocaml` so that, instead of having a single channel vector in the *pre* and *post* conditions, we can have a sequence of channel vectors, and we use lenses to *focus* on a channel vector at a particular *slot*. If we do not require delegation or interleaving, then the length of the sequence is one and the monadic

Dynamic	Static (monadic)
<pre>let s = send s#role_q#m v in e let s = send s#role_q#m s' in e match receive s#role_p with `m₁(x,s) -> e₁ `m₂(s',s) -> e₂ close s</pre>	<pre>let%lin #s_i = send s_i (fun x -> x#role_q#m) v in e let%lin #s_i = deleg_send s_i (fun x -> x#role_q#m) s_j in e match%lin receive s_i (fun x->#role_p) with `m₁(x,#s_i) -> e₁ `m₂(#s_j,#s_i) -> e₂ (delegation) close s_i</pre>

■ **Figure 14** OCaml API for MPST with Dynamic (a) and Static (b) linearity checks.

operations always update the first element of the sequence. In particular, as in [27], if a channel is delegated i.e sent through another channel, that slot (index) of the sequence is updated to `unit`, marking it as consumed.

The `ocaml-mpst` API, for static linearity checking, is given in Fig. 14(b), where s_i , and s_j in delegation, denote *lenses* pointing at i -th and j -th slot in the monad. The binary channels in the channel vector, used within the monadic primitives `send` and `receive`, are of the types given in Fig. 13(b). Functions `send` and `receive` both take (1) a lens s_i pointing to a channel vector; and (2) a selector function which extracts, from the channel vector at index s_i , a channel (`'v data, 't1`) `out` for output and `'a inp` for input. Type `data` denotes unrestricted (non-linear) payload types, whose values are matched against ordinary variables. The result of the monadic primitives is returned as a value of either type `'t lin` for output or `'a lin` for input, which is matched by `match%lin` or `let%lin`, ensuring the channels (and payloads, in case of delegation) are used linearly. A `lin` type must be matched against *lens-pattern* prefixed by `#`. Note that, `linocaml` overrides the `let` syntax and `#` pattern, in the way that `let%lin #si=exp` updates the index s_i , in the sequence of channel vectors, with the value returned from `exp`.

To realise session delegation, we have implemented a separate monadic primitive, `deleg_send si (fun x->x#p#1) sj`, presented in Fig. 14(b). The primitive extracts the channel vector at position s_i and then updates the channel vector at position s_j . As a result, the slot for s_j is returned and used in further communication, the slot s_i is updated to `unit`. An example program that uses `ocaml-mpst` static API is given in Fig. 4(b).

6 Evaluation

We evaluate our framework in terms of run-time performance (§ 6.1) and applications (§ 6.2, § 6.3). We compare the performance of `ocaml-mpst` with programs written in a continuation-passing-style (following the encoding presented in [53]) and untyped implementations (Bare-OCaml) that utilise popular communication libraries. In summary, `ocaml-mpst` has negligible overhead in comparison with *unsafe* implementations (Bare-OCaml), and CPS-style implementations. We demonstrate the applicability of `ocaml-mpst` by implementing a lot of use cases. In § 6.3, we show the implementation of the OAuth protocol, which is the first application of session types over `http`.

6.1 Performance

The runtime overhead of `ocaml-mpst` stems from the implementation of channel vectors, more specifically: (1) extracting a channel from an OCaml object when performing a communication action, and (2) either (2.1) dynamic linearity checks or (2.2) more closures introduced by the usage of a slot monad for static checking.

Our library is parameterised on the underlying communication transport. We evaluate its performance in case of synchronous, asynchronous and distributed transports. Specifically, we use the following communication libraries:

- (1) **ev**: OCaml’s standard **Event** channels which implements channels shared among POSIX-threads;
- (2) **lwt**: Streams between *lightweight-threads* [56], which are more efficient for I/O-intensive application in general, and broadly-accepted by the OCaml communities, and
- (3) **ipc**: UNIX pipes distributed over UNIX processes.

Note that **ev** is synchronous, while the other two are asynchronous. Also, due to current OCaml limitation, POSIX-threads in a process cannot run simultaneously in parallel, which particularly affects the overall performance of (1). As OCaml garbage collector is not a concurrent GC, only a single OCaml thread is allowed to manipulate the heap, which in general limits the overall performance of multi-threaded programs written in OCaml. For (3), we generate a single pipe for each pair of processes, and maintain a mapping between a local channel and its respective dedicated UNIX pipe. In addition, we also implement an optimised variant of `ocaml-mpst` in the case of **lwt**, denoted as **lwt-single** in Fig. 15; it reuses a single stream among different payload types, instead of using different channels for types. In particular, we cast a payload to its required payload type utilising `Obj.magic`, as proposed and examined by [40, 26]. Our benchmarks are generalisable because each microbenchmark exhibits the worst-case scenario for its potential source of overhead.

We compare implementations, written using (1) `ocaml-mpst` static API, (2) `ocaml-mpst` dynamic API, (3) a Bare-OCaml implementation using untyped channels as provided by the corresponding transport library, and (4) a CPS implementation, following the encoding in [47]. We have implemented the encoding manually such that a channel is created at each communication step, and passed as a continuation. Fig. 15 reports the results on three microbenchmarks.

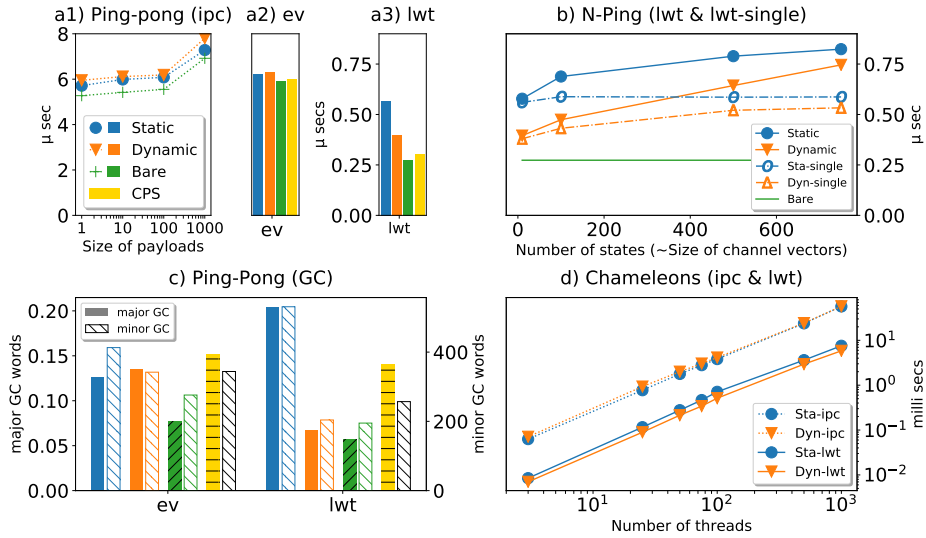
Setup. We use the native *ocamlopt* compiler of OCaml 4.08.0 with Flambda optimiser⁵. Our machine configurations are Intel Core i7-7700K CPU (4.20GHz, 4 cores), Ubuntu 17.10, Linux 4.13.0-46-generic, 16GB. We use `Core_bench`⁶, a popular benchmark framework in OCaml, which uses its built-in linear regression for estimating the reported costs. We repeat each microbenchmark for 10 seconds of quota where `Core_bench` takes hundreds of samples, each consists of up to 246705 runs of the targeted OCaml function, we obtain the average of execution time with fairly narrow 95% confidence interval.

Ping-pong. Ping-pong benchmark measures the execution time for completing a recursive protocol between two roles, which are repeatedly exchanging request-response messages of increasing size (measured in 16 bit integers). The example is communication intensive and exhibits no other cost apart from the (de)serialisation of values that happens in the **ipc** case, hence it demonstrates the pure overhead of channel extraction, dynamic checks and parameterised monads. In the case of a shared memory transports (**ev** and **lwt**), we report the results of a payload of one integer since the size of the message does not affect the running time.

The slowdown of `ocaml-mpst` is negligible (approx. 5% for Dynamic vs Bare-OCaml, and 13% for Static vs Bare-OCaml) when using either **ev**, Fig. 15 (a1), or **ipc**, Fig. 15(a2), as a transport, since the overhead cost is overshadowed by latency. The shared memory case

⁵ <https://caml.inria.fr/pub/docs/manual-ocaml/flambda.html>

⁶ https://blog.janestreet.com/core_bench-micro-benchmarking-for-ocaml/



■ **Figure 15** Runtime performance vs GC time performance.

using `lwt`, Fig. 15(a3), represents the worse case scenario for `ocaml-mpst` since it measures the pure overhead of the implementation of many interactions purely done on memory with minimal latency. The slowdown in the static version is expected [27] and reflects the cost of monadic closures, as the current implementation does not optimise them away. The linearity monad is implemented via a state monad [24], which incurs considerable overhead. The OCaml Flambda optimiser could remove more closures if we annotate the program with inlining specifications. The slowdown (although negligible) in comparison with CPS is surprising since we pre-generate all channels up-front, while the CPS-style implementation creates a channel at each interaction step. Our observation is that the compiler is optimised for handling large amounts of immutable values, while OCaml objects (utilised by the channel vector abstraction) are less efficient than normal records and variants.

Fig. 15 (c) reports on the memory consumption (in terms of words in the major and minor heap) for executing the protocol. Channel vectors with dynamic checking have approximately the same memory footprint as Bare-OCaml, and significantly less footprint when compared with a CPS implementation.

n-Ping. `n-Ping` is a protocol of increasing size, `nping` global combinator forming repeated composition of the communication combinators defined by $g_i = (a \rightarrow b) \text{ ping } @@ (b \rightarrow a) \text{ pong } @@ g_{i-1}$, $g_0 = \tau$ and $\text{nping} = \text{fix } (\text{fun } \tau \rightarrow g_n)$, where n corresponds to the number of `ping` and `pong` states. In contrast to Ping-Pong, this example generates a large number of channels and large channel vector objects, evaluating how well `ocaml-mpst` scales w.r.t the size of the channel vector structure. We show the results for transports `lwt` and `lwt-single` in Fig. 15 (b). The static version of `lwt-single` has a constant overhead from Bare-OCaml. Although the static checking implementation is in general slower, the relative overhead, in comparison with dynamic checking, decreases as the protocol length increases.

Chameleons. Chameleons protocol specifies that n roles (“chameleons”) connect to a central broker, who picks pairs and sends them their respective reference, so they can interact peer-to-peer. The example tests delegation (central broker sends a reference) and creation of

Example (role)	LoC	CT _(ms)	FM	Example (role)	LoC	CT _(ms)	FM
1. 2-Buyer [22]	15	45	✓	9. Game [47]	17	49	×
2. 3-Buyer [22]	21	47	✓	10. MapReduce [28]	5	33	×
3. Fibonacci [22]	8	38	×	11. Nqueen [28]	12	55	×
4. SAP-Negotiation [22]	17	46	×	12. Santa [38, 24]	14	42	×
5. Supplier Info [22]	50	85	✓	13. Sleeping Barber [22]	15	43	✓
6. SH [43, 22]	27	58	✓	14. SMTP [22]	54	124	×
7. Distributed Calc [22]	12	41	×	15. OAuth	26	60	✓
8. Travel Agency [22]	16	66	✓	16. DNS	11	57	×

■ **Figure 16** Implemented Use cases (LoC: Lines of code, CT: Compiling Time, FM: Full merge).

many concurrent sessions (peer-to-peer interaction of chameleons). The results reported in Fig. 15 (d) show that the implementation of delegation with static linearity checking scales as well as its dynamic counterpart. The cost of linearity (monadic closures) is less than the cost of dynamic checks for many concurrent sessions over `lwt` transport.

6.2 Use Cases

We demonstrate the expressiveness and applicability of `ocaml-mpst` by specifying and implementing protocols for a range of applications, listed in Fig. 16. We draw the examples from three categories of benchmarks: (1) *session benchmarks* (examples 1-9), which are gathered from the session types literature; (2) *concurrent algorithms* from the Savina benchmark suit [28] (examples 10-13); and (3) *application protocols* (examples 14-16), which focus on well-established protocols that demonstrate interoperability between `ocaml-mpst` implemented programs and existing client/servers. For each use case we report on Lines of Code (LoC) of global combinators and the compilation time (CT reported in milliseconds). We also report if the example requires full-merge [13] (FM) – a well-formedness condition on global protocols that is not supported in [47], but supported in `ocaml-mpst`.

Examples 1-9 are gathered from the official Scribble test suite⁷ [52], and we have converted Scribble protocols to global protocol combinators. Examples 10-13 are concurrent algorithms and are parametric on the number of roles (n). To realise the scatter-gather pattern required in the examples, we have added two new constructs, `scatter` and `gather`, which correspond to a subset of the parameterised role extension for MPST protocols [9].

To test the applicability of `ocaml-mpst` to real-world protocols we have specified, using global combinators, a core subset of three Internet protocols (examples 14-16), namely the Simple Mail Transfer Protocol (SMTP), the Domain Network System (DNS) protocol and the OAuth protocol. Using the `ocaml-mpst` APIs, it was straightforward to implement compliant clients in OCaml that interoperate with popular servers. In particular, we have implemented an SMTP client that interoperates with the Microsoft exchange server and sends an e-mail, an OAuth authorisation service that connects to a Facebook server and authenticates a client, and a DNS client and a server, which are implemented on top of a popular DNS library in OCaml (`ocaml-dns`). Note that DNS has sessions, as the DNS protocol has an ID field to discriminate sessions; and a request forwarding in the DNS protocol involves more than two participants (i.e. servers).

⁷ <https://github.com/scribble/scribble-java>

```

1 let fb_oauth =
2   (c -!-> s) (get "/start_oauth") @@
3   (s -?-> c) _302 @@ (* 302: HTTP redirect *)
4   (c -!-> a) (get "/login_form") @@
5   (a -?-> c) _200 @@
6   (c -!-> a) (post "/auth") @@
7   choice_at a (to_c success_or_fail)
8   (a, (a -?-> c) (_200_success ...)) @@
9   (c -!-> s) (success is_ok "/callback") @@
10  (s -!-> a) (get "/access_token") @@
11  (a -?-> s) _200 @@
12  (s -?-> c) _200 @@
13  finish)
14 (a, (a -?-> c) (_200_fail ...)) @@
15 (c -!-> s) (fail is_fail "/callback") @@
16 (s -?-> c) _200 @@
17 finish)

18 let fb_acceptor = H.start_server 8080 "/mpst-oauth"
19 let rec facebook_oauth_consumer () =
20   let ch = get_ch s fb_oauth in
21   let sid = string_of_int (Random.int ()) in
22   let conn = fb_acceptor sid in
23   let `get(_, ch) = receive (ch conn)#role_C in
24   let redir_url = fb_redirect_url sid "/callback" in
25   let ch = send ch#role_C#_302 redir_url in
26   let conn = fb_acceptor sid in
27   let ch = match receive (ch conn)#role_C with
28     | `success(_,ch) ->
29     let conn_p = H.http_connector
30       "https://graph.facebook.com/v2.11/oauth" in
31     let ch = send (ch conn_p)#role_A#get [] in
32     let `_200(auinfo,ch) = receive ch#role_A in
33     send ch#role_C#_200 "auth succeeded"
34     | `fail(_,ch) -> send ch#role_C#_200 "auth failed"
35   in close ch; facebook_oauth_consumer ()

```

■ **Figure 17** Global Combinators and Local Implementations for OAuth (excerpt).

6.3 Session Types over HTTP: Implementing OAuth

In this section, we discuss more details about `ocaml-mpst` implementation of OAuth⁸, which is an Internet standard for authentication. OAuth is commonly used as a way for Internet users to grant websites or applications access to their information on other websites but without giving them the passwords by providing a specific authorisation flow. Fig. 17 shows the specification of the global combinator, along with an implementation for the authorisation server. We have specified a subset of the protocol, which includes establishing a secure connection and conducting the main authentication transaction. Using OAuth as an example, we also discuss practically motivated extensions, *explicit connection handling* akin to the one in [23], to the core global combinators. We present that a common pattern when HTTP is used as an underlying transport.

Extension for handling stateless protocols. The protocol has a very similar structure to the `oAuth` protocol, presented in § 2. However, the original OAuth protocol is realised over a RESTful API, which means that every session interaction is either an HTTP request or an HTTP response. To handle HTTP connections, we have implemented a thin wrapper around an HTTP library, `Cohttp`⁹, and we make HTTP actions explicit in the protocol by proposing two new global combinators, *connection establishing* combinator (`-!->`) and *disconnection* combinator (`-?->`). Session types represent the types of the communication channel after a session (a TCP connection in the general case) has been established. Since RESTful protocols, realised over HTTP transport, are stateless, a connection is “established” at every HTTP Request. We explicitly encode this behaviour by replacing the `->` combinator that denotes that one role is sending to another, with two new combinators. The combinator `-!->` means establishing a connection and piggybacking a message, while `-?->` denotes piggybacking a message and disconnect. This simple extension allows us to faithfully encode HTTP Request and HTTP Response. For example, `a-!->b` requires that role `a` connects on an HTTP port to `b` and then `a` sends a message to `b`, hence implementing HTTP Response; on the other hand `a-?->b` specifies an HTTP Response.

⁸ <https://oauth.net/2/>

⁹ <https://github.com/mirage/ocaml-cohttp>

Implementation. The global combinator `fb_oauth` is given in Fig. 17 (a). As before, the protocol consists of three parties, a service `s`, a client `c`, and an authorisation server `a`. First, `c` connects to `s` via a relative path `"/start_oauth"` (Line 2). Then `s` redirects `c` to `a` using HTTP redirect code `_302` (Line 3). As a result the client sees a login form at `"/login_form"` (Lines 4-5), where they enter their credentials (Line 6). Based on the validity of the credentials received by `c`, `a` sends `_200_success` (Line 8) or `_200_fail`. If the credentials are valid, `c` proceeds and connects to `s` on path `"/callback"` (Line 9), requesting to get access to a secure page. The service `s` then retrieves an *access token* from `a` on URL `"/access_token"` (Lines 10-11), and navigates the client to an authorised page, finishing the session (Lines 12-13). If the credentials are not valid, the client reports the failure to `s` (Lines 15-16), and the session ends (Line 17).

The server role of `fb_oauth` is faithfully implemented in Lines 18-35 which provides an OAuth application utilising Facebook's authentication service. Line 18 starts a thread which listens on a port 8080 for connections. Essentially it starts a web service at an absolute URL `"/mpst-oauth"` (i.e. relative URLs like `"/callback"` are mapped to `"https://.../mpst-oauth/callback"`). The recursive function `facebook_oauth_consumer` starting from Line 19 is the main event loop for `s`. Line 20 extracts a channel vector from the global combinator `fb_oauth`, of which type is propagated to the rest of the code. Then it generates a session id via a random number generator (`Random.int ()`) (Line 21), and waits for an HTTP request from a client on `fb_acceptor` (Line 22). When a client connects, the connection is bound to the variable `conn` associated with the pre-generated session id. Note that the channel vector expects a connection since no connection has been set for the client yet. Here, the connection is supplied to the channel vector via function application (`ch conn`). On Line 24, expression `(fb_redirect_url sid "/callback")` prepares a redirect URL to an authentication page of a Facebook Provider (`https://www.facebook.com/dialog/oauth`) After sending back (HTTP Response) the redirect url to the client with `_302` label (Line 25), the connection is implicitly closed by the library. Note that we do not need to supply a connection to the channel vector on Line 25; because a connection already exists, we have already received an HTTP request from the user and Line 25 simply performs HTTP response. The next lines proceed as expected following the protocol, with the only subtlety that we thread the connection object in subsequent send/receive calls.

The full source code of the benchmark protocols and applications and the raw data are available from the project repository.

7 Related Work

We summarise the most closely related works on session-based languages or multiparty protocol implementations. See [52] for recent surveys on theory and implementations.

The work most closely related to ours is [47], which implements multiparty session interactions over binary channels in Scala built on an encoding of a multiparty session calculus to the π -calculus. The encoding relies on *linear decomposition* of channels, which is defined in terms of *partial projection*. Partial projection is restrictive, and rules out many protocols presented in this paper. For example, it gives an undefined behaviour for role `c` and `s` for protocols `oAuth2` and `oAuth3` in Fig. 3. Programs in [47] have to be written in a continuation passing style where a fresh channel is created at each communication step. In addition, the ordering of communications across separate channels is not preserved in the implementation, e.g. sending a `login` and receiving a `password` in the protocol `oAuth` is decomposed to two separate elements which are not causally related. This problem is

mitigated by providing an external protocol description language, Scribble [50], and its API generation tool, that links each protocol state using a call-chaining API [22]. The linear usage of channels is checked at runtime.

An alternative way to realise multiparty session communications over binary channels is using an orchestrator – an intermediary process that forwards the communication between interacting parties. The work [6] suggests addition of a medium process to relay the communication and recover the ordering of communication actions, while the work [7] adds annotations that permit processes to communicate directly without centralised control, resembling a proxy process on each side. Both of the above works are purely theoretical.

Among multiparty session types implementations, several works exploit the equivalence between local session types and communicating automata to generate session types APIs for mainstream programming languages (e.g., Java [22, 30], Go [9], F# [47]). Each state from state automata is implemented as a class, or in the case of [30], as a type state. To ensure safety, state automata have to be derived from the same global specification. All of the works in this category use the Scribble toolchain to generate the state classes from a global specification. Unlike our framework, a local type is not inferred automatically and the subtyping relation is limited since typing is nominal and is constrained by the fixed subclassing relation between the classes that represent the states. All of these implementations also detect linearity violations at runtime, and offer no static alternative.

In the setting of binary session types, [27] propose an OCaml library, which uses a slot monad to manipulate binary session channels. Our encoding of global combinators to simply-typed binary channels enable the reuse of the techniques presented in [27], e.g. for delegations and enforcement of linearity of channels.

FuSe [41] is another library for session programming in OCaml. It supports a runtime mechanism for linearity violations, as well as a monadic API for a single session without delegation. The implementation of FuSe is based on the encoding of binary session-typed process into the linear π -calculus, proposed by [12]. The work [48] also implements this encoding in Scala, and the work [47] extends the encoding and implementations to the multiparty session types (as discussed in the first paragraph).

Several Haskell-based works [43, 39, 31] exploit its richer typing system to statically enforce linearity with various expressiveness/usability trade-offs based on their session types embedding strategy. These works depend on type-level features in Haskell, and are not directly applicable to OCaml. A detailed overview of the different trade-off between these implementations in functional languages is given in Orchard and Yoshida’s chapter in [52]. Based on logically-inspired representation of session types, embedding higher-order binary session processes using contextual monads is studied in [54]. This work is purely theoretical.

Outside the area of session-based programming languages, various works study protocol-aware verification. Brady et al. [5] describe a discipline of protocol-aware programming in Idris, in which adherence of an implementation to a protocol is ensured by the host language dependent type system. Similarly, [51] proposes a programming logic, implemented in the theorem prover Coq, for reasoning on protocol states. A more lightweight verification approach is developed in [1] for a set of protocol combinators, capturing patterns for distributed communication. However, the verification is done only at runtime. The work [8] presents a global language for describing choreographies and a global execution model where the program is written in a global language, and then automatically projected using code generation to executable processes (in the style of BPMN). All of the above works either develop a new language or are built upon powerful dependently-typed host languages (Coq, Idris). Our aim is to utilise the MPST framework for specification and verification of distributed protocols, proposing a type-level treatment of protocols which relies solely on existing language features.

8 Conclusion and Future Work

In this work, we present a library for programming multiparty protocols in OCaml, which ensures *safe* multiparty communication over binary I/O channels. The key ingredient of our work is the notion of global combinators – a term-level representation of global types, that automatically derive channel vectors – a data structure of nested binary channels. We present two APIs for programming with channel vectors, a monadic API that enables static verification of linearity of channel usage, and one that checks channel usage at runtime. OCaml is intensively used for system programming among several groups and companies in both industry and academia [35, 3, 32, 33, 34, 15, 10, 44]. We plan to apply `ocaml-mpst` to such real-world applications.

We formalise a type-checking algorithm for global protocols, and a sound derivation of channel vectors, which, we believe, are applicable beyond OCaml. In particular, TypeScript is a promising candidate as it is equipped with a structural type system akin to the one presented in our paper.

To our best knowledge, this is the first work to enable MPST protocols to be written, verified, and implemented in a single (general-purpose) programming language and the first implementation framework of statically verified MPST programs. By combining protocol-based specifications, static linearity checks and structural typing, we allow one to implement communication programs that are extensible and type safe by design.

References

- 1 Kristoffer Just Arndal Andersen and Ilya Sergey. Distributed protocol combinators. In *Practical Aspects of Declarative Languages - 21th International Symposium, PADL 2019, Lisbon, Portugal, January 14-15, 2019, Proceedings*, volume 11372 of *Lecture Notes in Computer Science*, pages 169–186. Springer, 2019. doi:10.1007/978-3-030-05998-9_11.
- 2 Robert Atkey. Parameterized Notions of Computation. *Journal of Functional Programming*, 19(3-4):335–376, 2009. doi:10.1017/S095679680900728X.
- 3 Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Timothy L. Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles 2003, SOSP 2003, Bolton Landing, NY, USA, October 19-22, 2003*, pages 164–177, 2003. doi:10.1145/945445.945462.
- 4 Frédéric Bour, Thomas Refis, and Gabriel Scherer. Merlin: a language server for ocaml (experience report). *PACMPL*, 2(ICFP):103:1–103:15, 2018. doi:10.1145/3236798.
- 5 Edwin Charles Brady. Type driven development of concurrent communicating systems. *Computer Science*, 18(3), July 2017. doi:10.7494/csci.2017.18.3.1413.
- 6 Luís Caires and Jorge A. Pérez. Multiparty session types within a canonical binary theory, and beyond. In *Formal Techniques for Distributed Objects, Components, and Systems - 36th IFIP WG 6.1 International Conference, FORTE 2016, Held as Part of the 11th International Federated Conference on Distributed Computing Techniques, DisCoTec 2016, Heraklion, Crete, Greece, June 6-9, 2016, Proceedings*, volume 9688 of *Lecture Notes in Computer Science*, pages 74–95. Springer, 2016. doi:10.1007/978-3-319-39570-8_6.
- 7 Marco Carbone, Sam Lindley, Fabrizio Montesi, Carsten Schürmann, and Philip Wadler. Coherence generalises duality: A logical explanation of multiparty session types. In *27th International Conference on Concurrency Theory, CONCUR 2016, August 23-26, 2016, Québec City, Canada*, volume 59 of *LIPICs*, pages 33:1–33:15. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2016. doi:10.4230/LIPICs.CONCUR.2016.33.
- 8 Marco Carbone and Fabrizio Montesi. Deadlock-freedom-by-design: multiparty asynchronous global programming. In *The 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '13, Rome, Italy - January 23 - 25, 2013*, pages 263–274. ACM, 2013. doi:10.1145/2429069.2429101.

- 9 David Castro, Raymond Hu, Sung-Shik Jongmans, Nicholas Ng, and Nobuko Yoshida. Distributed Programming Using Role Parametric Session Types in Go. In *46th ACM SIGPLAN Symposium on Principles of Programming Languages*, volume 3, pages 29:1–29:30. ACM, 2019.
- 10 Patrick Chanezon. Docker for mac and windows beta: the simplest way to use docker on your laptop, March 2016. URL: <https://blog.docker.com/2016/03/docker-for-mac-windows-beta/>.
- 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*, volume 9104 of *LNCS*, pages 146–178. Springer, 2015. doi:10.1007/978-3-319-18941-3_4.
- 12 Ornela Dardha, Elena Giachino, and Davide Sangiorgi. Session Types Revisited. In *PPDP '12: Proceedings of the 14th Symposium on Principles and Practice of Declarative Programming*, pages 139–150, New York, NY, USA, 2012. ACM. doi:10.1145/2370776.2370794.
- 13 Pierre-Malo Deniérou and Nobuko Yoshida. Multiparty session types meet communicating automata. In *ESOP*, volume 7211 of *LNCS*, pages 194–213. Springer, 2012. doi:10.1007/978-3-642-28869-2.
- 14 Pierre-Malo Deniérou and Nobuko Yoshida. Multiparty compatibility in communicating automata: Characterisation and synthesis of global session types. In *ICALP*, volume 7966 of *LNCS*, pages 174–186. Springer, 2013.
- 15 Fabrice Le Fessant. MLDonkey, 2002. <http://mldonkey.sourceforge.net/>.
- 16 J. Nathan Foster, Michael B. Greenwald, Jonathan T. Moore, Benjamin C. Pierce, and Alan Schmitt. Combinators for bidirectional tree transformations: A linguistic approach to the view-update problem. *ACM Trans. Program. Lang. Syst.*, 29(3):17, 2007. doi:10.1145/1232420.1232424.
- 17 Silvia Ghilezan, Svetlana Jaksic, Jovanka Pantovic, Alceste Scalas, and Nobuko Yoshida. Precise subtyping for synchronous multiparty sessions. *J. Log. Algebr. Meth. Program.*, 104:127–173, 2019. doi:10.1016/j.jlamp.2018.12.002.
- 18 Dick Hardt. The OAuth 2.0 Authorization Framework. RFC 6749, October 2012. doi:10.17487/RFC6749.
- 19 Robert Harper and Benjamin C. Pierce. A record calculus based on symmetric concatenation. In *Conference Record of the Eighteenth Annual ACM Symposium on Principles of Programming Languages, Orlando, Florida, USA, January 21-23, 19x91*, pages 131–142, 1991. doi:10.1145/99583.99603.
- 20 Kohei Honda, Nobuko Yoshida, and Marco Carbone. Multiparty asynchronous session types. In *POPL'08*, pages 273–284. ACM, 2008.
- 21 Kohei Honda, Nobuko Yoshida, and Marco Carbone. Multiparty asynchronous session types. *J. ACM*, 63(1):9:1–9:67, 2016. doi:10.1145/2827695.
- 22 Raymond Hu and Nobuko Yoshida. Hybrid session verification through endpoint API generation. In *FASE*, volume 9633 of *LNCS*, pages 401–418. Springer, 2016. doi:10.1007/978-3-662-49665-7_24.
- 23 Raymond Hu and Nobuko Yoshida. Explicit connection actions in multiparty session types. In *FASE*, volume 10202 of *LNCS*, pages 116–133, 2017. doi:10.1007/978-3-662-54494-5_7.
- 24 Keigo Imai and Jacques Garrigue. Lightweight linearly-typed programming with lenses and monads. *Journal of Information Processing*, 27:431–444, 2019. doi:10.2197/ipsjjip.27.431.
- 25 Keigo Imai, Rumyana Neykova, Nobuko Yoshida, and Shoji Yuen. Multiparty session programming with global protocol combinators, 2020. arXiv:2005.06333.
- 26 Keigo Imai, Nobuko Yoshida, and Shoji Yuen. Session-ocaml: A session-based library with polarities and lenses. In *COORDINATION*, volume 10319 of *LNCS*, pages 99–118. Springer, 2017. doi:10.1007/978-3-319-59746-1_6.
- 27 Keigo Imai, Nobuko Yoshida, and Shoji Yuen. Session-ocaml: a Session-based Library with Polarities and Lenses. *Sci. Comput. Program.*, 172:135–159, 2018. doi:10.1016/j.scico.2018.08.005.

- 28 Shams Imam and Vivek Sarkar. Savina - An Actor Benchmark Suite: Enabling Empirical Evaluation of Actor Libraries. In *AGERE*, pages 67–80. ACM, 2014.
- 29 Oleg Kiselyov. Simple variable-state monad, December 2006. Mailing list message. <http://www.haskell.org/pipermail/haskell/2006-December/018917.html>.
- 30 Dimitrios Kouzapas, Ornela Dardha, Roly Perera, and Simon J. Gay. Typechecking protocols with Mungo and StMungo. In *PPDP*, pages 146–159, 2016. doi:10.1145/2967973.2968595.
- 31 Sam Lindley and J. Garrett Morris. Embedding Session Types in Haskell. In *Haskell 2016: Proceedings of the 9th International Symposium on Haskell*, pages 133–145. ACM, 2016. doi:10.1145/2976002.2976018.
- 32 Anil Madhavapeddy. Xen and the art of OCaml. In *Commercial Uses of Functional Programming (CUFP)*, September 2008.
- 33 Anil Madhavapeddy and David J. Scott. Unikernels: the rise of the virtual library operating system. *Commun. ACM*, 57(1):61–69, 2014. doi:10.1145/2541883.2541895.
- 34 Dirk Merkel. Docker: Lightweight linux containers for consistent development and deployment. *Linux Journal*, 2014(239), March 2014. URL: <http://dl.acm.org/citation.cfm?id=2600239.2600241>.
- 35 Yaron Minsky. OCaml for the Masses. *Commun. ACM*, 54(11):53–58, 2011. doi:10.1145/2018396.2018413.
- 36 Rumyana Neykova, Raymond Hu, Nobuko Yoshida, and Fahd Abdeljallal. A session type provider: compile-time API generation of distributed protocols with refinements in f#. In *Proceedings of the 27th International Conference on Compiler Construction, CC 2018, February 24-25, 2018, Vienna, Austria*, pages 128–138. ACM, 2018. doi:10.1145/3178372.3179495.
- 37 Rumyana Neykova and Nobuko Yoshida. Featherweight Scribble. In *Models, Languages, and Tools for Concurrent and Distributed Programming - Essays Dedicated to Rocco De Nicola on the Occasion of His 65th Birthday*, pages 236–259, 2019. doi:10.1007/978-3-030-21485-2_14.
- 38 Nick Benton. Jingle Bells: Solving the Santa Claus Problem in Polyphonic C#, 2003. Available at <https://www.microsoft.com/en-us/research/wp-content/uploads/2016/02/santa.pdf>.
- 39 Dominic Orchard and Nobuko Yoshida. Effects as sessions, sessions as effects. In *POPL 2016: 43th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 568–581. ACM, 2016. doi:10.1145/2837614.2837634.
- 40 Luca Padovani. A Simple Library Implementation of Binary Sessions. *Journal of Functional Programming*, 27:e4, 2016.
- 41 Luca Padovani. Context-free session type inference. *ACM Trans. Program. Lang. Syst.*, 41(2):9:1–9:37, 2019. doi:10.1145/3229062.
- 42 Matthew Pickering, Jeremy Gibbons, and Nicolas Wu. Profunctor Optics: Modular Data Accessors. *The Art, Science, and Engineering of Programming*, 1(2):Article 7, 2017. doi:10.22152/programming-journal.org/2017/1/7.
- 43 Riccardo Pucella and Jesse A. Tov. Haskell session types with (almost) no class. In *Haskell’08*, pages 25–36, New York, NY, USA, 2008. ACM. doi:10.1145/1411286.1411290.
- 44 Gabriel Radanne, Jérôme Vouillon, and Vincent Balat. Eliom: A core ML language for tierless web programming. In *Programming Languages and Systems - 14th Asian Symposium, APLAS 2016, Hanoi, Vietnam, November 21-23, 2016, Proceedings*, pages 377–397, 2016. doi:10.1007/978-3-319-47958-3_20.
- 45 John H. Reppy. Concurrent ML: Design, Application and Semantics. In *Functional Programming, Concurrency, Simulation and Automated Reasoning: International Lecture Series 1991-1992, McMaster University, Hamilton, Ontario, Canada*, pages 165–198, 1993. doi:10.1007/3-540-56883-2_10.
- 46 Davide Sangiorgi and David Walker. *The π -Calculus: a Theory of Mobile Processes*. Cambridge University Press, 2001.
- 47 Alceste Scalas, Ornela Dardha, Raymond Hu, and Nobuko Yoshida. A Linear Decomposition of Multiparty Sessions for Safe Distributed Programming. In *ECOOP*, 2017. doi:10.4230/LIPIcs.ECOOP.2017.24.

- 48 Alceste Scalas and Nobuko Yoshida. Lightweight session programming in scala. In *ECOOP*, volume 56 of *LIPICs*, pages 21:1–21:28, 2016. doi:10.4230/LIPICs.ECOOP.2016.21.
- 49 Alceste Scalas and Nobuko Yoshida. Less Is More: Multiparty Session Types Revisited. In *46th ACM SIGPLAN Symposium on Principles of Programming Languages*, pages 1–29. ACM, 2019.
- 50 Scribble home page, 2019. URL: <http://www.scribble.org>.
- 51 Ilya Sergey, James R. Wilcox, and Zachary Tatlock. Programming and proving with distributed protocols. *PACMPL*, 2(POPL):28:1–28:30, 2018. doi:10.1145/3158116.
- 52 António Ravara Simon Gay, editor. *Behavioural Types: from Theory to Tools*. River Publisher, 2017. URL: https://www.riverpublishers.com/research_details.php?book_id=439.
- 53 The Scala Development Team. The Scala Programming Language, 2004. URL: <http://scala.epfl.ch/index.html>.
- 54 Bernardo Toninho, Luís Caires, and Frank Pfenning. Higher-order processes, functions, and sessions: A monadic integration. In *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 350–369. Springer, 2013. doi:10.1007/978-3-642-37036-6_20.
- 55 Jesse A. Tov and Riccardo Pucella. Stateful contracts for affine types. In Andrew D. Gordon, editor, *Programming Languages and Systems, 19th European Symposium on Programming, ESOP 2010, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2010, Paphos, Cyprus, March 20-28, 2010. Proceedings*, volume 6012 of *Lecture Notes in Computer Science*, pages 550–569. Springer, 2010. doi:10.1007/978-3-642-11957-6_29.
- 56 Jérôme Vouillon. Lwt: a cooperative thread library. In *Proceedings of the ACM Workshop on ML*, pages 3–12. ACM, 2008. Available at <https://github.com/ocsigen/lwt>. doi:10.1145/1411304.1411307.
- 57 Mitchell Wand. Type inference for record concatenation and multiple inheritance. *Inf. Comput.*, 93(1):1–15, 1991. doi:10.1016/0890-5401(91)90050-C.

Designing with Static Capabilities and Effects: Use, Mention, and Invariants

Colin S. Gordon 

Department of Computer Science, Drexel University, Philadelphia, PA, USA
csgordon@drexel.edu

Abstract

Capabilities (whether object or reference capabilities) are fundamentally tools to restrict effects. Thus static capabilities (object or reference) and effect systems take different technical machinery to the same core problem of statically restricting or reasoning about effects in programs. Any time two approaches can in principle address the same sets of problems, it becomes important to understand the trade-offs between the approaches, how these trade-offs might interact with the problem at hand.

Experts who have worked in these areas tend to find the trade-offs somewhat obvious, having considered them in context before. However, this kind of design discussion is often written down only implicitly as comparison between two approaches for a specific program reasoning problem, rather than as a discussion of general trade-offs between general classes of techniques. As a result, it is not uncommon to set out to solve a problem with one technique, only to find the other better-suited.

We discuss the trade-offs between static capabilities (specifically reference capabilities) and effect systems, articulating the challenges each approach tends to have in isolation, and how these are sometimes mitigated. We also put our discussion in context, by appealing to examples of how these trade-offs were considered in the course of developing prior systems in the area. Along the way, we highlight how seemingly-minor aspects of type systems – weakening/framing and the mere existence of type contexts – play a subtle role in the efficacy of these systems.

2012 ACM Subject Classification Theory of computation → Type theory; Software and its engineering → Language features

Keywords and phrases Effect systems, reference capabilities, object capabilities

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2020.10

Category Pearl

Acknowledgements Many thanks are due to the audience at the OCAP 2018 workshop where these ideas were initially presented, and to the ECOOP 2020 reviewers, for helpful feedback on the ideas, presentation, and paper itself.

1 Introduction

Capabilities are a classic idea [35, 34] with intuitive appeal: explicitly tie possession of certain entities to the ability to perform certain actions, so by bounding the flow of those entities one can restrict the possible actions of a program or program component [45]. Much of the work in this area centers the notion of *object capabilities*, where capabilities control access to objects (in the OO sense), and capabilities are realized as object references: a program fragment cannot modify or invoke operations of an object it cannot reference. This immediately grants a way to control mutation of objects, and by tying external calls to specific objects, also extends to controlling externally-visible behaviors as well. For example, by associating all file operations with a particular *object* – not a globally accessible library call – developers may tightly control which code can access those operations by restricting how widely the file operations object is distributed. Intuitively, capabilities act as permission to do things, and the absence of capabilities acts as a lack of permission. It is also possible to delegate partial access to an object’s operations using proxy objects [8, 61, 60] or through capabilities acting



© Colin S. Gordon;

licensed under Creative Commons License CC-BY

34th European Conference on Object-Oriented Programming (ECOOP 2020).

Editors: Robert Hirschfeld and Tobias Pape; Article No. 10; pp. 10:1–10:25

Leibniz International Proceedings in Informatics



LIPICs Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

10:2 Designing with Static Capabilities and Effects: Use, Mention, and Invariants

as handles to trusted mediators [35]. However, doing this kind of reasoning statically is also appealing, because it incurs no runtime performance overhead when delegating or mediating access.

As a result, there is now a rich body of work on statically checked capabilities. Once static reasoning is employed, the kinds of restrictions proxies and mediators permit in object capability systems may not require new dynamic objects exposing different sets of operations. One of the most well-developed bodies of work on static capabilities uses *reference capabilities*, which associate different permissions¹ to individual references in a program, in contrast to the object capability view that all references to an object are equal and restrictions stem from using different objects with fewer or modified operations available. Thus, different references to the same object – distinguished by a type system or static analysis, but not the runtime system – may permit programs holding them different abilities to affect object state or invoke certain operations. Most reference capability systems are type systems where reference types come equipped with a type qualifier [21] corresponding to certain permissions (and in some cases, invariants or assumptions about aliases). Capability-based reasoning is supported by checking the types flowing into a given program context.

Different variations of reference capabilities have been employed to solve a wide array of programming problems. Systems with read-only reference capabilities [59, 66, 16, 49, 11] restrict some references to read-only access to their referents – even when aliases exist that can be used to mutate the referent – which is useful for preventing a wide variety of accidental mutations, from expressing that a method treats its arguments as deeply read-only to controlling consequences of representation exposure [15]. This can be combined nicely with object immutability [65], as all references to an immutable object are read-only. Transitive versions have been used to ensure data race freedom in Microsoft prototypes [29] and the Pony programming language [12]: if two threads only shared (transitively) read-only references, no data races can exist between them.² They have also been used to infer method purity [33, 32]: if a method accepts only (transitively) read-only inputs (including the receiver), it has no externally-visible side effects.³ In other contexts, program behavior can be constrained by building more fine-grained capabilities that grant not only all-or-none permission to mutate, but can grant permission for only certain kinds of mutation, and can therefore enforce nuanced invariants by restricting which capabilities can coexist for the same resource [27, 25, 28, 43, 44, 5, 6].

Each of these systems makes critical use of the original motivation for capabilities: by restricting what flows into certain parts of the program, one can provide guarantees about what that part may do – without precisely examining the semantics of its internals.

For the problems mentioned above, there also exist effect systems to statically check the same high level concepts (e.g., for data race freedom [1], or purity [54]). In contrast to capability systems which reason externally in terms of what capabilities flow into code, effect systems are a class of type system extensions that analyze program behavior⁴ by (to a first

¹ Typically reference capabilities are distinguished statically, though a dynamic interpretation is possible.

² This is simplifying away some of the substructural aspects of these type systems, which all make use of forms of uniqueness to *also* support partitioning mutable data between threads, or in Encore’s case [6] restrict conflicting accesses to atomic synchronization primitives.

³ This sets aside extensionally-observable effects, such as allocating memory or triggering GC.

⁴ By effect systems, we mean the sort of type system extension that reasons about bounds on program behavior as part of the type judgment, in the sense of the work on FX that originally coined the term “effect system” [40, 24]. This stands in contrast to *denotational* approaches which attempt to assign meaning to effects, often by way of monads or some extension thereof, following Moggi [46]. Filinski [20] offers an excellent discussion of the distinction. Readers familiar with algebraic effects should note that

approximation) performing a bottom-up analysis of what interesting actions might occur, based on (typically) a join semilattice of effects: primitive or external actions of interest are typed as having particular specific effects representing their behavior of interest, and larger expressions' effects are computed by taking the least upper bound of subexpressions' effects. This raises a fundamental question: when considering a static reasoning approach to a problem, how do we recognize which approach is likely to be better suited? Comparisons between these systems in the literature tend to focus on the low-level expressive distinctions between systems for a particular problem domain (e.g., System A accepts this data race free program rejected by System B), or the relative complexity of the type rules (as a proxy for usability). While the core trade-offs are there if one looks carefully, the broader issue of contrasting the trade-offs between these *classes* of solutions has received little explicit attention.

In our personal experience, it is not uncommon to set out to build a capability-based system, only to find effect systems more suitable to the task at hand – and sometimes the reverse. People experienced with both effect systems and capability systems – whether as designers, or users – likely find this unsurprising. But to the best of our knowledge, there is essentially no discussion in the literature on how system designers chose one approach over the other; systems are presented as complete and finished designs, then evaluated against other finished designs – the design process is lost. Having a record of these trade-offs and design questions would be useful, both for shared understanding and especially for newcomers to static capabilities or effects. Part of this requires identifying and developing terminology for aspects of these trade-offs.

In this expository paper we articulate some of the core trade-offs between these static reasoning approaches, and how these trade-offs are moderated in important ways by some of the most humble of reasoning principles in type systems: weakening and the use of type contexts. We also explain how the trade-offs have affected the design of several reference capability systems and effect systems we have worked on. We expect that little of what we say would be surprising to those who have worked on both static reference capability systems and type-and-effect systems; we view our contributions as primarily giving clear explicit exposition to these trade-offs that are generally left implicit in the literature, and putting those trade-offs in context by providing some extra information on the design evolution of a couple effect systems and capability systems. Our hope is that newcomers to these areas and their intersection, or outsiders looking in, will find these distinctions helpful.

2 Capabilities, Use, and Mention

One of the original goals for capability-based design is to reason about the effect of some code by reasoning about the capabilities it is provided [45, Ch. 8] – a long-standing practice based on the notion that capabilities essentially grant permission to cause effects, though until relatively recently [13, 38, 39] the exact relationship between static capabilities and static effects was left implicit.⁵ However, the pure form of this approach – that the set of capabilities provided is used to give an upper bound on an expression's effect – has limitations we have not seen crisply articulated in a general way before.

much work on algebraic effects involves *both* varieties: handlers define the semantics of user-defined effects, but a restrictive type system of the sort we discuss ensures all effect operations are invoked in the context of an enclosing handler.

⁵ This despite being noted as a relevant question (in other terms) much earlier [21].

Before we can precisely articulate the limitations of a kind of reasoning, let us first clarify exactly what kind of reasoning we mean. We will ground discussion primarily in systems using read-only references to control side effects [65, 66, 59] to control side effects. We will draw on a prototype dialect of C# [29] that used these and with context bounding to both control interference between threads, and also to strengthen typing assumptions. These ideas were later generalized in Pony [12], so much of our discussion applies fairly directly there as well, and less precisely to a range of earlier [65, 66, 59], contemporaneous [32, 33], and later [23] systems. This line of work, focusing on read-only references devoid any meaning besides mutability restrictions, historically used the term *reference immutability* to describe the relevant techniques (as in, an object was immutable *through a particular (read-only) reference*). This was partly to distinguish itself from other techniques with read-only references as part of systems that captured more design intent, like owner-as-modifier ownership types [49, 11, 10] or universe types [17, 16]. To date, the particular sort of capability-bounding we discuss below has only been explored for systems in the so-called reference immutability family.

In most of these systems, the specific capability appears as a type qualifier [21] modifying a basic object (class) type. In the C# dialect we discuss, there were four reference capabilities: **isolated** (externally unique [31]), **readable** (transitively read-only), **writable** (mutable), and **immutable** (transitively immutable, in the sense that the immediate referent and all objects reachable from it are permanently immutable). **readable** and **immutable** may be used for reading fields, may not be used for mutation, and may not be used to *obtain* references usable for mutation. For example, reading a **writable**-declared field through a **readable** reference would produce only a **readable** reference – e.g., iterating over the elements of a **readable** `List<writable Foo>` would work through a series of **readable** `Foo` instances. In the case of an immutable referent, stronger results appear because everything reachable via an **immutable** reference is an immutable object: iterating over a **immutable** `List<writable Foo>` would see **immutable** `Foo` instances.

Setting aside some subtleties related to uniqueness, the simplest embodiment of using context bounds to reason about effects in these systems is the parallel composition type rule from the C# dialect, present in an analogous form in related systems [12, 23]:

$$\frac{\text{T-PAR} \quad \text{NoWritable}(\Gamma_1, \Gamma_2) \quad \Gamma_1 \vdash C_1 \dashv \Gamma'_1 \quad \Gamma_2 \vdash C_2 \dashv \Gamma'_2}{\Gamma_1, \Gamma_2 \vdash C_1 \parallel C_2 \dashv \Gamma'_1, \Gamma'_2}$$

The rule above simply says that as long as two thread bodies (C_1 and C_2) require no **writable** variables in their inputs, it is safe (data-race-free) to run them in parallel (and the output of the flow-sensitive typing judgment combines per-thread outputs in the obvious way). This works because in order for a data race to occur, one thread would need write access to an object the other thread could reach. Prohibiting **writable** references from entering either thread guarantees this cannot happen: any object reachable from both threads would be truly immutable, or both threads would have only **readable** references to it. Crucially, this reasoning is sound because the C# dialect – like Pony [12], Encore [6], and L42 [23] – prohibited global mutable state, providing a form of capability safety [45]: assurance that reasoning in terms of only capabilities directly entering an expression was sound, because there were no *ambient* capabilities (those that can be obtained by any code at any time). This rule also partitions some mutable state between threads, by splitting up **isolated** references.

Another, slightly less traditional form of effect-bounding is the notion of *recovery*, first proposed by Gordon et al. [29], adapted by Clebsch et al. [12] for use in Pony, and later extended for better flexibility by Giannini et al. [23]. Again, we will demonstrate with the simplest rule, one of two given for the C# dialect:

$$\frac{\text{T-RECOVERIMM} \quad \text{IsolatedOrImmutable}(\Gamma) \quad \text{IsolatedOrImmutable}(\Gamma') \quad \Gamma \vdash C \dashv \Gamma', x : \text{readable } D}{\Gamma \vdash C \dashv \Gamma', x : \text{immutable } D}$$

This rule says that if all inputs and all but one output x of a command are **isolated** or **immutable**, and the other output x is **readable**, then it is safe to recover the *stronger immutable* capability for x – stronger because **immutable** D is a subtype of **readable** D , making this a kind of statically safe downcast. Intuitively this handles either the case that x already points to immutable data, or the case that it points to mutable data that is unreachable except via x and can therefore be “frozen” to immutable. The restrictions on Γ and Γ' ensure x doesn’t alias mutable state, since the lack of ambient capabilities means x must point to an input (all **immutable** or **isolated**– and therefore freezable) or something allocated within C (which cannot escape via C ’s inputs). As with the concurrency rule, the soundness of this relies fundamentally on the fact that weak permissions on the inputs imposes strong restrictions on the code’s behavior (plus the prohibition on ambient authority / global mutable state).

This rule makes it possible to write code that handles some number of immutable or externally-unique data with code that was not written with strict immutability (as opposed to **readable**) in mind:

```
readable T RandomChoice(readable T a, readable T b) { ... }
...
{x,y:immutable T}
z = RandomChoice(x,y);
{x,y:immutable T,z:readable T}
{x,y,z:immutable T}
```

The code above passes two immutable references to **RandomChoice**, which assumes it simply returns a **readable** reference. But with the recovery rule above, the result (z) can be recovered as immutable – it must either be pointer-equal to x or y , or a new T allocated inside the method, which is therefore not aliased elsewhere, and can be converted to immutable.

In the C# prototype, and now Pony, this kind of reasoning has worked out well. But why, and where would it break?

2.1 The Gap Between Capability Bounds and Effects: Use-Mention Distinction

These kinds of reasoning could be done using explicit effect systems [40, 24]. But what does that gain us? As is known [42, 13], an explicit effect system requires a system that cares about the details of the code being analyzed, which can require complex types and effects [1] (we see examples in Section 3). So concretely, what can effect systems offer that capability-based reasoning struggles with?

The key point of departure between this capability-bound-based reasoning and a general effect system is what we will refer to as a kind of *use-mention distinction*. In philosophy and linguistics, logical fallacies and confusion are known to arise from conflating *use* of a thing with mere *mention* of a thing [47, 14]. Reasoning about an expression’s effect using only

10:6 Designing with Static Capabilities and Effects: Use, Mention, and Invariants

$$\begin{array}{c}
 \text{HeapWrite} \\
 | \\
 \text{NoHeapWrite}
 \end{array}
 \quad
 \frac{\text{T-VARASSIGN} \quad x \in \Gamma \quad y \in \Gamma}{\Gamma \vdash x = y : \text{unit} \mid \text{NoHeapWrite}}
 \quad
 \frac{\text{T-FIELDASSIGN} \quad \Gamma(x) = \text{writable } C \quad \tau f \in \text{Fields}(C) \quad \Gamma \vdash e : \tau \mid \chi}{\Gamma \vdash x.f := e : \text{unit} \mid \text{HeapWrite}}$$

■ **Figure 1** An overly-simple effect system (excerpt) that could parallelize a local assignment of writable variables.

the capabilities it has access to inherently performs the same kind of conflation: possessing authority means only that the code has the *ability* to use it, not that it necessarily does. This seems to be anecdotally understood among designers of static capability systems, but rarely discussed. To the best of our knowledge, this paper is the first to explicitly call out and name this trade-off.

Consider the following in the C# reference immutability dialect:

```

{x:readable T,y:writable T,z:writable T}
y = z;
/* actual concurrent work with x, but not y or z */
{x:readable T,y:writable T,z:writable T}

```

The single local variable assignment is enough to prevent parallelization (as the full body of a thread) via T-PAR even though it will never cause a data race in the heap, because `y` and `z` are typed as `writable` but T-PAR forbids `writable` references in a thread’s initial type environment. Every sound type system will reject some semantically valid code, but this example seems particularly innocuous.

Consider for contrast the overly-simple effect system and rules in Figure 1. There are two effects, `NoHeapWrite` \sqsubseteq `HeapWrite`. Every primitive expression that is not a heap write is given effect `NoHeapWrite` (notably, variable assignments), the expression that performs a write into the heap has effect `HeapWrite`, and compound expressions’ effects are simply the least upper bound of the subexpressions’ effects. This way, any expression containing *any* heap write would be given effect `HeapWrite`. The line of code above would have effect `NoHeapWrite` – which implies it could be parallelized without a data race.

Clearly this toy example will not scale up to real imperative programs (it likely won’t handle the “actual concurrent work” assumed in the example), but it is still instructive because it already highlights the use-mention distinction: the code above *mentions* the writable references, but does not *use* them in a way relevant to the property of interest (heap mutation).

Thus the fact that capability-bound-based reasoning does not inspect the internals of an expression is a strength in that it reduces complexity, but also a weakness because it inherently loses precision.

It is worth briefly noting that there exist reference capability systems where some references are usable only for comparing object identity, and not for actually causing effects, as in Pony’s `tag` permission [12], or much earlier in Boyland et al.’s unifying framework for reference capabilities [3]. Such restricted references remain useful for code without permissions to invoke operations implemented in code with permissions [51]. In Pony, such references are permitted to enter recover blocks, because they do not affect the capability-bounded reasoning: they are references that do not act as capabilities (for the mutation effects addressed by Pony).

Other kinds of related, but different, distinctions have appeared in the literature on object capabilities. Miller [45] and later Drossopoulou et al. [18] distinguish permission as direct access to an object (to invoke its methods), and authority as the ability to cause effects on an object. Drossopoulou et al. [18] showed that in general such notions of permission do not imply authority (a direct reference to an object with only pure methods grants permission, but not authority, over that object), and authority does not imply permission (invoking a method may cause mutations to an object the caller lacks direct access to). This distinction is further related to distinguishing permission (or authority) in a given program state from the permission (or authority) obtainable via further execution, either of a specific program, of any program adhering to some behavioral specification, or of any possible program. The use-mention distinction somewhat resembles the distinction between eventual permission (for a given program) and behavioral permission (roughly, for all programs preserving the typing discipline, which due to types controlling permissions is also similar to Miller’s notion of a topology-of-permissions based bound on authority), which also touches upon the distinction between what a program might actually do based on its code versus what it may have (or obtain) authority to do ignoring the details of the particular program. We propose the use-mention distinction not to supplant such analyses of capability systems, but specifically to distinguish the loss of precision capability-bound reasoning suffers in comparison to effect systems.

2.2 Working Around Use-Mention Conflation

That the Microsoft C# prototype was used to write an entire operating system kernel [19] and Pony is used in industry suggest that at least sometimes, this use-mention distinction is not critical. Certainly, few developers wish to parallelize a local variable assignment alone.

There are also ways to work around this limitation when it otherwise might arise. Reference capability type systems typically include weakening (or in the C# case, framing) type rules, that allow variables that are *not even mentioned* to be temporarily set aside and ignored, allowing capability-based reasoning to be applied more locally.

$$\text{T-WEAKENING} \frac{\Gamma \vdash e : \tau}{\Delta, \Gamma \vdash e : \tau} \quad \text{T-FRAME} \frac{\Gamma \vdash C \dashv \Gamma'}{\Delta, \Gamma \vdash C \dashv \Delta, \Gamma'}$$

Both rules simply state that if an expression or command is well-typed with certain variables, then it remains well-typed (with the same type) in the presence of additional variables. Often this is enough to side-step conflation of use and mention: operations like the problematic local variable write above can frequently be refactored to a separate part of the program (e.g., before or after introducing concurrency), and this is arguably better coding style anyways.

Consider a variation on the recovery example:

```

{ x, y : immutable T, b : writable U }
  { x, y : immutable T }
  { x, y : immutable T }
  z = RandomChoice(x, y);
  { x, y : immutable T, z : readable T }
  { x, y, z : immutable T }
{ x, y, z : immutable T, b : writable U }

```

} T-RECOVERIMM
} T-FRAME

This is the same code as the previous recovery example, but type-checked with an additional variable `b` in scope with a `writable` permission. The initial type environment would fail the `IsolatedOrImmutable` check in `T-RECOVERIMM` because `b` is `writable`. However

framing away the extra `writable` variable that is not needed in the recovery region (i.e., instantiating T-FRAME with $\Delta = b : \text{writable } U$) allows recovery to be used with an environment containing only `x` and `y`, both `immutable`. Thus while context-bounding risks losing precision due to the inability to distinguish use and mention, this weakness is tempered in a subtle way by the most humble of type system rules. An under-appreciated aspect of these rules in type theories is that they imply the “extra” variables in Δ are *definitely not used* by the expression at hand.⁶

It is known that removing structural rules like weakening leads to very different type theories (substructural type theories [62]), but we believe we are the first to remark upon this interplay between *weakening* and the precision of context-bounded reasoning as a general phenomenon, rather than simply exploiting it. Unique, linear, and affine capabilities all typically rely on restricting a different structural rule (*contraction*) that permits multiple uses of the same variable (including in the aforementioned read-only reference systems).

Another more unique use of structural constraints and capabilities is the work of Giannini et al. [23], who extend the expressivity of the C# dialect and Pony’s recovery. Those languages require strict lexical nesting of recovery blocks, which can make some sophisticated uses of recovery difficult to write. Giannini et al. modify the structure of contexts to track multiple sets of variables for recovery simultaneously (keeping them separated), allowing a typing derivation to switch between active sets for different expressions, without any particular nesting order. They motivate this extension from a very pragmatic point of view, but their enhancement is essentially enriching contexts with additional structure typical of a substructural logic or type system, with their new rules playing the role of novel structural rules that permute the context to swap active and “inactive” portions. They noticed an interplay between structural rules and reference capabilities in a particular context, but did not highlight it as a general issue. Still, the general issue and their result suggest deeper investigation of the interactions between capabilities and structural rules is warranted.

2.3 The Limits of Workarounds

Ultimately, even with the subtle benefits of weakening, the question of whether the use-mention distinction is important depends on the specific problem at hand. For safe parallelism and method purity, the past few years have strongly suggested that the use-mention distinction is not a serious problem. Since capability-based reasoning about those effects is usually powerful enough, it is usually preferable to a full effect system due to its comparative simplicity (we see the alternative in Section 3).

Contrast this against another problem: preventing any thread other than the distinguished UI event loop thread from directly updating objects representing the UI – considered an error in most UI frameworks, often resulting in program termination if a program violates this discipline. In prior work, we proposed an effect system [26] that prevented such errors. Like the reference capability examples mentioned earlier, this has also seen adoption in industry (through Stein et al.’s clever extensions [55]), offering some evidence that this was a good design decision.

A key part of the work was distinguishing which objects had UI-related methods and which objects did not. This was delineated in the type system using a type qualifier – the same type of machinery used to manage reference capabilities – but the actual analysis relied

⁶ This is slightly surprising in contrast to separation logic, where the equivalent framing rule is (rightly) viewed as a powerful reasoning principle [53].

```

1  final @UI JLabel label = ...;
2  new Thread() { // ← Captures label reference
3      public void run() { // ← label reference in scope
4          // do really slow computation
5          Display.asyncExec(new @UI Runnable() { // ← Captures label reference again
6              public void run() {
7                  label.setText("Complete!"); // ← Use on UI thread
8              }
9          });
10 }
11 }.run();

```

■ **Figure 2** UI event handler code spawning a background thread that sends code back to the UI thread.

on an effect system. Because the qualifiers could be interpreted as capabilities (a thread cannot access UI elements if it holds no references to UI objects), a plausible alternative to an effect system would have been to use a context restriction on code that ran on background threads (those that should not update the UI directly): forbid them access to UI-related objects, by a rule similar to the safe parallelism rule shown earlier. This work was carried out shortly after work on the C# dialect, in parallel with a related reference capability system [27] refining the notion of read-only references. As a result, we considered this approach during the design of what became an effect system.

But the challenge is this: the details of how background threads notify the UI of completed work. Consider this typical sequence of steps in a user interface. When the user clicks a button, an event handler is triggered on the UI event loop thread to handle the input. If the work to be done is expensive, then rather than blocking the UI thread, the handler offloads work to a background thread. Running work on the background thread will allow the UI to respond to other inputs while the work is ongoing. But once the work is done the display must be updated with the results. Background threads are forbidden from directly updating the UI themselves, for a variety of reasons discussed elsewhere [26]. So when the work is completed, the code executing on the *background thread* must somehow trigger an update to occur on the UI thread to indicate completion and/or display the results.

In all current UI frameworks, this occurs by permitting the background thread to hold (mention) a reference to UI elements, and send them in a closure to the UI thread – which then executes the code, using the reference to update the UI. Figure 2 gives a concrete example of this. The `JLabel` on line 1 in Figure 2 is a UI element that should only be used on the UI thread. But the background thread code (the `Thread.run` implementation starting on line 3) holds a reference to the label through the expensive work, which is then passed back to the UI thread inside a `Runnable`, whose body (line 7) is then safely invoked from the UI thread. Preventing the flow of any `@UI` object references into background threads would reject this code – and essentially all code written for existing UI libraries. In this case, an effect system was required to distinguish use and mention.

The use-mention distinction also arises in a second form for this problem: existing code mixes methods that should run on background threads in the same classes as methods that must run on the UI thread. Arguably this could be recast as a granularity issue – splitting capabilities into those granting UI method rights and those not granting UI method rights, following the compatible aliasing approach we discuss later, could work. But in that case it leads to capability types that are more complex than the effects – the capabilities would need to track sets of permitted methods, while there are only two effects in the solution (plus effect variables for effect polymorphism): `@SafeEffect` \sqsubseteq `@UIEffect`.

2.3.1 Counterarguments

One possible objection to the above is that the problem above may be avoidable through use of different abstraction principles, such as defining the `Runnable` above in a context with the `JLabel` in scope, applying some variant of an anti-frame rule [52] – a formalization of information hiding, in this case encapsulating a capability inside the `Runnable` – to encapsulate the reference, and then defining the thread separately such that it cannot even (directly) mention the `JLabel`. However, this alone simply inverts the problem with use-mention distinctions: rather than treating mention as use, it hides both! To ensure the background thread does not call the `run()` method that accesses the label, it is necessary to prevent use (calling). To allow the functionality it is necessary to still allow the thread to pass the `Runnable` to `Display.asyncExec`. To permit one without the other requires another distinction of use and mention – which we would argue, is an effect system. In addition, such an approach would also prohibit background thread code from, for example, preparing a list of objects to update on the UI thread, which inherently requires the ability to mention the UI object references for storage.

A potentially stronger counterargument might stem from claiming that the difficulty with context bounding above stems from conflating capabilities with references, as all reference capability systems do. This conflation means that capabilities can be stored in the heap. In contrast, static capabilities divorced from data may permit additional separation: the UI thread might possess a static capability that it keeps, and UI-sensitive operations (methods) should require (and return) this unique capability. This does make it impossible to invoke a UI operation on a background thread! However, we would argue that this is essentially an effect system: `@UIEffect` can be read as marking methods that require and return the hypothetical separate capability. We are not alone in this view.

Walker et al. [63] give a translation from the region calculus of Tofte and Talpin [57, 58] to a calculus of static capabilities (independent from values), and note that for this class of capabilities the distinction is in some ways a subjective difference between analyzing the behavior of code (as an effect system or monadic approach might) or dictating up front what the permissible actions are (the capability view).

More recent work on capability-based effect systems similarly takes the explicit view that capabilities grant permission to cause effects, leading to systems that restrict effects by restricting the flow of capabilities. Liu et al. [38, 39] propose distinguishing *stoic* functions as those that do not capture capabilities (directly or indirectly), and obtain stoic functions purely by capability-bounded reasoning: all functions are initially typed as possibly capturing, and a function that is well-typed in a context with no capabilities (or capability-capturing closures) can be downcast to a stoic function type (akin to recovery), which means any effects of the function then appear explicitly in its signature as capability arguments, akin to a latent effect (taking the capability as an argument does not oblige the function to use it directly). Careful use of stoic functions could be used to ensure background thread code does not capture the hypothetical UI capability, making the distinction between the two effects of interest equivalent to whether or not code accepts the UI capability as an argument. Liu et al. refer to program changes to pass capabilities instead of capturing them as “making their effects explicit.” Osvald et al. [50] explicitly equate the capabilities required for a method with method effects, following Marino and Millstein’s generic effect framework [41] that explicitly formulates effects as sets of capabilities.

3 Effects, Naming, and Invariants

Given the fact that effect systems can handle the use-mention distinction, why would we ever use only capabilities to bound behaviors in a static system? The main *technical* reason to choose capabilities is that they permit reasoning about effects for code that is not inspected, as in precompiled library code when retrofitting a type system, or dynamically loaded code. But in the case that all code is compiled with a tool performing the same analysis (supporting separate compilation), this advantage is less important. Why would we choose capabilities over effects in this case?

The answer is informal and subjective: simplicity. Simplicity when capabilities are adequate in practice is a compelling answer for many reasonable people. But the previous section gave an example where an effect system not only handled the use-mention distinction, but was also simpler than a plausible capability-based approach. It turns out, simplicity often favors the other direction. Effect systems excel at reasoning about the behavior of individual sections of code – but not at reasoning about the behavior of all code at the same time on specific shared objects with many different names. In short, effect systems struggle to retain simplicity while enforcing invariants, particularly when they must relate multiple names to multiple entities (which is necessary to ensure multiple uses are similar).

3.1 A Thought Experiment: Replacing Reference Immutability with Effects

Consider, as we did, designing an effect system that accepts precisely the same programs as a reference immutability system. For simplicity let us consider ReIm [33], which has only mutable and transitively read-only references – no uniqueness, and no absolute immutability. The type rules for this system are fairly straightforward: they extend the standard class-based object-oriented type system rules to include the qualifiers in the subtyping relation, and beyond this administrative “plumbing” the main changes are the same one common to all deep reference immutability type systems:

- The rule for type checking field writes requires the reference to the modified object to be **writable**.
- The rule for field reads ensures that if the base object reference used for a field read is **readable**, then so is the result, regardless of the permission in the field declaration.

As a consequence of these rules, for a program to follow a path through the heap to perform a write, every reference traversed along that path (local variable and field type alike) must be **writable**.

An effect system with the same precision in terms of *which references are used (transitively) for mutation* is quite complex. Assuming all local variables are let-bound (i.e., final, and cannot be rebound) for simplicity, indicating that a variable was used directly for writing is straightforward:

$$\frac{\Gamma(x) = T \quad \tau \in \text{Fields}(T) \quad \Gamma \vdash e : \tau \mid \chi}{\Gamma \vdash x.f := e : U \mid \{wr(x)\} \cup \chi}$$

This rule simply takes the type τ and effect χ of the right hand side, and adds to it an effect indicating the base reference x was used for writing. The challenge arises when reconciling external and internal variables. Consider:

let $x = e_1$ in e_2

If e_2 contains a write through x , then e_2 's effect should include $wr(x)$, indicating that x is used as if it were mutable. But outside the body of this `let`, x is meaningless⁷ – what it refers to depends on e_1 , and in general may refer to one of several objects (e.g., if e_1 involves a conditional or heap dereference). A sound effect system would need to take any effects on x and conservatively assume they could occur for any of the objects e_1 may evaluate to. But this then requires the effect system to reason about may-alias relationships – possible, but tricky, since this in turn requires naming sets of objects in the heap in a precise manner. Essentially, an effect system approach collects aliasing and use information and propagates it outwards to be reasoned about wholesale. For a transitive reference immutability system like ReIm, this information would also need to track origin information: it is possible that x itself may never be used for writing in e_2 , but some other reference, obtained by reading through x could be – and in that case, x would need to be indicated as usable for (transitive) write access as well.

One could consider extending this experiment to more nuanced systems of read-only references. We considered such an experiment ourselves after working on the UI threading effect system, trying to build a precise effect system analogue of the C# reference immutability system; the naming and usage information for an effect system approach to that language seems to grow even faster than for ReIm. The same extrapolation applies to related systems like Pony [12] and L42 [23].

In this case using an effect system seems highly undesirable, and prone to significant complexity. What changed from the UI threading effect system? In this thought experiment, we considered a system where access paths through the heap are important, and object identity is important. For the UI threading case, neither of those are true. A diligent student of the literature on effect systems might point out the similarities between the considerations for `let`-binding above and the `letregion` construct in calculi for region-based memory management [56, 58]. These calculi have effect systems with similar read and write effects on a per-*region* basis, rather than per object, and the effects are read and write behaviors to specific regions. This separation from naming individual objects or tracking access paths is a substantial simplification. The case of a region name being limited to a specific lexical scope also arises for `letregion`, but there the region that is undefined outside that scope simply doesn't exist – nor do any data or types that might depend on it – because the binding construct is also the (de)allocation construct, and typing rules for `letregion` forbid the appearance of the bound (then deallocated) region in the construct's result type. Object- and reference capability systems tend to be used for situations involving one or both of these features that lead to more complex effects – object identity and heap paths.

3.2 Global Invariants via Local Capabilities

Capabilities, on the other hand, allow this kind of reasoning to be handled purely locally, usually without naming issues or explicit tracking of access paths. Type contexts, along with the field type look-ups typical in type systems for OO languages excel at identifying sets of objects used similarly, because they actually *force* sets of objects to be used similarly – the type system will statically ensure that all values dynamically bound to a certain variable (or field) are used at the same type. When absolute similarity is problematic, polymorphism over types or permissions is possible [16, 29, 36]. This is important because these points of the system – variable and field types – already conflate types of different objects in standard

⁷ Or worse, means something else if it was shadowing another x .

type systems. So tying capabilities to variable and field types essentially enforces a kind of invariant: it conflates capabilities in the same places a traditional type system already conflates basic types. As a result, this leads to little additional friction for developers already using a typed language. Effect systems such as the hypothetical effect version of reference immutability must somehow reconstruct this sort of conflation that comes for free when the effects are restricted *by* the type context.

Static reference capability systems of recent years also all carry a notion of *compatibility* between references/capabilities. In many static reference capability systems, each reference permission comes with not only restrictions on how it is used, but restrictions on how *aliases* are used. These systems maintain a global invariant that for any two aliases, the permissions granted via one reference are a subset of the interference assumed by the other, in both directions. The early papers on rely-guarantee references [27], rely-guarantee protocols [43], and Pony [12] give particularly thorough accounts of this. This notion of compatibility between aliases is imposed any time references are duplicated, and in the case of systems like Kappa [5], joined as well.

Preserving compatibility between aliases can also be done locally, without name binding issues. In each case, one type A may be split into two others B and C if:

- B and C 's combined capabilities do not exceed A 's original capabilities for modification, and
- B (resp. C) assumes at least as much interference as A assumed
- B (resp. C) assumes at least as much interference as C 's (resp. B 's) capabilities provide.

As a concrete example, consider the rely and guarantee components of a rely-guarantee reference [27, 28], which specify binary relations constraining what modifications that reference may be used for (the guarantee) and what its aliases may be used for (the rely). A reference of type $\text{ref}\{\mathbb{N} > 5\}[\leq, =]$ refers to a natural number strictly greater than 5, assumes aliases may increment the number (any time an alias modifies the stored value, the old value must be \leq the new value, and typing may *rely* on this fact), and may only be used for reading (or non-modifying writes; new values must be $=$ the old value, and the type system must *guarantee* uses obey this restriction). This may be split into two copies of itself (it is *reflexively splittable*), because none of the three (original and the two split copies) permits writes, but all would tolerate increments through aliases. Moreover, because the predicate on the referent (that it is greater than 5) is preserved by the guarantee (equality), this check on reference splitting ensures the predicate will be preserved by *all* possible references, with only point-wise checks every time a new alias is created. In contrast, a reference of type $\text{ref}\{\mathbb{N} > 5\}[=, \leq]$ may be used for incrementing, but assumes all aliases are read-only. So it may not be duplicated naively: each copy would assume it was the *only* reference that could be used for increments. This permits some very granular reasoning about side effects, without a full effect system (though again, not distinguishing mention and use).

As one could imagine, extending our thought experiment of a purely effect system replacement for ReIm to a system like this would produce very complex effects, adding constraints from these binary relations into effects dealing with naming and aliasing. By enforcing this restriction on duplicating references, the type system can ensure the value stored in that reference remains greater than 5 without explicitly tracking where the aliases go or when they are used.

In the “reference immutability” family of read-only reference type systems [29, 12, 32, 33, 65, 66, 59], compatibility typically requires no special care – the shape of the permission subtyping relationships already ensures any duplication preserves compatibility (setting

aside unique references). `readable` and `writable` references assume aliases may mutate the referent, and while `immutable` references assume no aliases may mutate it, they also do not grant permission for mutation, so duplication is not problematic.

In other systems, the changes remain relatively local following the general argument above. Rely-guarantee references [27, 28] use a notion of type splitting, $\Gamma \vdash \tau \prec \tau' * \tau''$ to check that when a value (particularly one containing references) is duplicated, it can be split into compatible types τ' and τ'' . It generally recursively checks splitting, bottoming out at the reference splitting rule, which looks somewhat complex but merely formalizes the three aspects of compatibility above (plus preservation of predicates):

$$\text{REF-*} \frac{\Gamma \vdash \text{ref}\{b \mid \phi'\}[R', G'] \quad \Gamma \vdash \text{ref}\{b \mid \phi''\}[R'', G''] \quad \emptyset \subset \llbracket G' \rrbracket \subseteq \llbracket R'' \rrbracket \quad \emptyset \subset \llbracket G'' \rrbracket \subseteq \llbracket R' \rrbracket \quad \llbracket G' \rrbracket \cup \llbracket G'' \rrbracket \subseteq \llbracket G \rrbracket \quad \llbracket R \rrbracket \subseteq \llbracket R' \rrbracket \quad \llbracket R \rrbracket \subseteq \llbracket R'' \rrbracket}{\Gamma \vdash \text{ref}\{b \mid \phi\}[R, G] \prec \text{ref}\{b \mid \phi'\}[R', G'] * \text{ref}\{b \mid \phi''\}[R'', G'']}$$

This formalizes splitting type A into types B and C ($\Gamma \vdash A \prec B * C$) when all are rely-guarantee references. Beyond checking that the new types B and C are well-formed, it checks that B and C 's combined capabilities (guarantees) do not exceed A 's ($G' \cup G'' \subseteq G$), that B assumes at least as much interference as A ($R \subseteq R'$), and that B tolerates interference from C ($G'' \subseteq R'$) (plus the symmetric checks on C).

This splitting check is inserted into a couple obvious locations in static reference capability systems, wherever new aliases may be created – variable reads, memory reads, and parameter passing. Rely-guarantee protocols [43, 44] do a form of model checking to check compatibility in the same places. Kappa [5] has a similar notion of packing and unpacking composite capabilities. Maintaining this compatibility invariant with only local checks means that the concurrent versions of these systems [44, 5, 28] no longer require explicit bounding checks for concurrency – simply splitting well-formed type contexts (and certain assumptions about the granularity of interleaving) is sufficient for safety. And because the combined permissions of the two new references cannot grant more authority than the original's permissions, any invariant enforced by the original is enforced by both new references as well.⁸ Typestate managed via permission [48, 22] has analogous checks.

This discussion, however, is abstracted from concrete use cases. And it is worth asking whether some particular aspects of reference immutability, particularly the transitive variants, might make the problem worse than it could be (though we didn't get that far above).

3.3 Invariants for JavaScript, Instead of Effects

We previously encountered the challenges involved in maintaining global invariants with effects when designing a type system to enable efficient ahead-of-time compilation of JavaScript [7]. The goal was to allow JavaScript to be run on embedded devices, faster than via an interpreter, but with lower memory footprint than a JavaScript JIT (which in addition to keeping the compiler in memory, keeps multiple versions of the code). The core idea behind the type system was to use types to rule out JavaScript behaviors that are especially difficult to optimize at compile time – those that would seem to require a JIT to execute efficiently – while permitting some of JavaScript's (in)famous flexibility that did not seriously interfere with compilation. JavaScript's semantics are full of cases that are difficult to compile efficiently ahead of time, but we will focus on one particularly tricky case that pushed the team towards capabilities.

⁸ In systems that permit recombining reference capabilities [5, 6, 44, 43], the new reference may grant more permissions than the two original pieces, but the system maintains that rejoining previously-split references never grants more authority than the original.

```

function F() {
  this.x = 0
}
F.prototype.inc = function() { this.x++; }
F.prototype.count = function() { return this.x; }
F.prototype.incAndCount = function() {
  this.inc();
  return this.count();
}
/* construct a new F instance, and increment its x field */
var f = new F(); // f.x == 0
f.inc(); // f.x == 1
/* add the field x to F.prototype */
F.prototype.inc();

```

■ **Figure 3** Violating fixed-object layout.

One aspect of JavaScript that makes it particularly difficult to optimize is the fact that object layouts are not fixed – fields may be added or removed dynamically. This means the typical approach to compiling field accesses in a language like Java or C – emitting a constant-time access to a statically-known offset from the object’s base pointer – does not work in general. Fortunately, a significant amount of JavaScript code is reasonably well-behaved and does not add fields once an object is fully initialized. But because normal JavaScript will silently create fields if a program writes to one that doesn’t exist, it is easy to do this unintentionally.

Consider the code in Figure 3. `F` is a (pre-ES6) constructor. Calling `new F()` allocates a new object, sets `F.prototype` as that object’s *prototype* (source of inherited properties), and executes the code of the function `F` with that new object as the receiver. In JavaScript, if a field is read on an object, but does not exist there, the runtime checks for that field in the object’s prototype. If it is there, it returns the value from the prototype. Otherwise the runtime checks the prototype’s prototype, and so on, until the field is found or there are no more prototypes. A field write, however, always writes to the immediate referent, and never consults the prototype chain. This makes subtle mistakes possible. The call to `f.inc()` increments the field `x` in `f` as expected; `inc` is found in the prototype object, invoked with `f` as the receiver, and the write in that method writes to `f`. The last line of Figure 3 invokes the method on the *prototype*, however, which is probably not supposed to have an `x` field at all. In standard JavaScript runtimes, this would run without error: reads of undefined fields return a special `undefined` value, which is coerced to a number (really, `NaN`) by addition, and the increment then writes to `f`, which will result in the runtime dynamically adding the field. But `F.prototype` is intended to be the equivalent of an abstract class – all methods, no data. For the purposes of ahead-of-time compilation, this would be a problem to avoid.

The heart of the problem above is that the `inc` method writes to `this.x`, and therefore should only be executable on objects that (should) have a field `x` before the call. The last line of code should then be rejected because it calls `inc` on an object missing required fields. The actual system design included many other issues, but this problem could be viewed as the defining challenge for the system: if all objects were guaranteed to have fixed object layout, then a runtime system incapable of dynamic field addition and removal could still preserve the original program semantics.

10:16 Designing with Static Capabilities and Effects: Use, Mention, and Invariants

Building a type system for a dynamic language essentially always requires structural types (i.e., record types with width subtyping [4]), which enumerate which fields were present in each object, leading to types like

$$\{x : \text{number}, y : \text{number}, m : () \rightarrow \text{number}\}$$

indicating two numeric fields and a method returning a number.⁹ Initial work on the project [9] also made clear a need to distinguish definitely-local fields (like `f.x`) that could be written safely, and possibly-inherited fields (like `f.inc`) – field accesses to the former can be compiled more efficiently than the latter. This leads to split object types of the form $\{r \mid w\}$, where r contains the types of *readable* fields known to be present somewhere (locally or inherited), and w contains the types of *writable* fields known to be present on the immediate referent.

We can explore another thought experiment, which is actually a reproduction of the original trajectory in designing this as an effect system, prior to correcting to a capability system. Initially, it appeared¹⁰ we should view the problem in terms of which fields of each object were accessed (in which ways) by each section of code. In hindsight, we can concisely state that the goal was to ensure each object had a fixed object layout, and that all references to each object collaboratively maintained that fixed layout as an invariant, as alluded to in the previous subsection. Both of these are correct points of view, but they lead to very different system designs.

3.3.1 The Effect System Approach

An early approach to handling the problem in Figure 3 was an effect system tracking which fields of the receiver were written by each method. In this case, the problematic call above is rejected by the draft rule T-MCALLSKETCH: `F.prototype`'s type does not include `x`, while `inc`'s effect would indicate it would write `this.x`.

$$\text{T-MCALLSKETCH} \quad \frac{\Gamma \vdash e : \{r \mid w\} \mid \chi \quad m : (\tau_1, \dots, \tau_n) \xrightarrow{\chi_m} \tau \in (r \cup w) \mid \chi_i \quad \forall i \in 1..n. \Gamma \vdash e_i : \tau_i \mid \chi_i \quad \chi_m \subseteq w}{\Gamma \vdash e.m(e_1 \dots e_n) : \tau \mid \chi \cup \bigcup_{i \in 1..n} (\chi_i)}$$

In particular, the final antecedent (the subset check) would fail.

Going even slightly beyond this example, however, quickly pushes this idea into unwieldy territory, because this requires tracking not only presence or absence of object modification as in the previous thought experiment, but also which parts of an object were modified. Objects also sometimes pass the receiver as an argument to methods of *other* objects (notice that if one of the parameters passed in T-MCALLSKETCH is the receiver, this – unsoundly – does not affect the overall effect). So to track the correct set of receiver field writes for a method containing `foo.bar(this)`, it becomes necessary to track which fields `foo.bar` actually writes to on its (initial) first argument – accounting for subsequent aliasing and transitive calls within `foo.bar` as well.

⁹ In this exposition, we will only consider methods, even though the full system supported functions as well.

¹⁰ What follows reflects a personal view of what appeared “obvious” at different points in time, and the actual design process the present author engaged in; we do not mean to suggest our coauthors were predisposed to the same mistakes.

But the trouble does not end there, as it did in the ReIm effect system thought experiment above. Reference immutability type systems (and reference capability systems in general) only articulate constraints on interface components – the receiver, method parameters, and return value – and need not explicitly describe internal behaviors, keeping the types relatively simple. These effects, however, expose internal implementation details of objects, like “private” field names. For examples like Figure 3 alone, this abstraction violation is merely uncomfortable. But it quickly becomes a technical problem.

Notice that instances of Figure 3’s \mathbf{F} implement a structural interface with methods to increment a counter and get its current value. Assuming the split object types outlined above, \mathbf{f} can be given a concise type:

$$\{inc : () \xrightarrow{x} () \mid x : \text{number}\}$$

This type says the object has (possibly-inherited) fields `inc` and `get`, and a local field `x`. If another object \mathbf{g} implements the same interface, but uses internal field name `y` to store its count, it would have type:

$$\{inc : () \xrightarrow{y} () \mid y : \text{number}\}$$

Now we have a problem: what are the effects of these methods in the least common supertype of these types, which we would need to store \mathbf{f} and \mathbf{g} in the same local variable or pass them to the same methods? The increment method’s effect mentions `x` in \mathbf{f} ’s type, while the effect of \mathbf{g} ’s increment method mentions `y`. The effects are incompatible.

Depth subtyping on mutable records is unsound in general, but the methods are in the read-only part of the object (since they are inherited), so depth subtyping is sound for them. This means that for the `inc` method, using subtyping to over-approximate the actual effect of each method is sound, so the least upper bound of the incrementing interfaces could then be:

$$\{inc : () \xrightarrow{x,y} () \mid \}$$

This combines width subtyping (which drops fields that do not exist in both objects) with depth subtyping on the read-only fields. This is a meaningful upper bound: the latent effect over-approximates both implementations’ effects. But `x` and `y` do not appear in this type, so checking that such an object contains all fields mentioned in the method effects in order to type-check a method invocation would fail – and in fact, neither object has *both* field `x` and field `y`.

We can resolve this, perhaps, by existentially quantifying over the particular field. But since this is a general issue of representing internal state, we must also abstract over the field’s type. And of course, there’s no requirement that two implementations of the same abstract interface use the same *number* of fields to store their state, leading to existential quantification over *rows* [64] – essentially fragments of object types¹¹:

$$\exists X :: \text{row}. \exists W :: \text{row}. \{inc : () \xrightarrow{\text{wr}(X)} () \mid W\}$$

This type essentially says the `inc` method modifies some set X of receiver fields, and existentially quantifies over locally-present fields.

¹¹ Rows were originally used as an alternative to bounded polymorphism in object or record calculi, such as $\forall X :: \text{row}. x \notin X \Rightarrow \{x : \text{number}, X\} \times \{x : \text{number}, X\} \rightarrow \{x : \text{number}, X\}$ as the type of a function that takes two objects with common fields including a field x , and returning whichever has the larger value in the field x . Rows are now also used in effect systems [37] in an analogous way, but this is orthogonal to our capabilities vs. effects discussion.

10:18 Designing with Static Capabilities and Effects: Use, Mention, and Invariants

But even this is not a complete solution! Now we can again store references to **f** and **g** in the same storage location by making different choices for the existentials, and now no longer leak information about the names of internal fields. But we haven't solved the original problem. We still need to know if the now-existentially-quantified row of fields written by the method is a subset of the fields actually present in the object in order to invoke the method. This information is not only lost by width subtyping and the abstraction of the existential, but the relationship between the row and other fields the object may contain is not captured by the type.

In the more concrete case of **f** and **g**, their common supertype again cannot explicitly mention the presence of **x** or **y**, since neither field is in both objects. This leads to further existential quantification, and bounding of row variables! To actually *invoke* `inc` through the abstract interface, we must know the written fields are a subset of the present fields. We can embed this information by using *bounded existentially quantified row variables*:

$$\exists W :: \text{row}. \exists X \subseteq W \{ \text{inc} : () \xrightarrow{\text{wr}(X)} () \mid W \}$$

But at the cost of some complexity, it seems this does offer a path to solve the original problem: each method may possibly write different subsets of local fields, and it seems if enough constraints are added, it should be possible to make the necessary connections to check that invoked methods only access fields that are actually present on the receiver.

Yet it is still not a complete solution. This path can handle the increment example. But to solve the original problem, two *additional and substantial* extensions are still required. First, there is a parallel problem with methods possibly *reading* fields that may not be present in the prototype chain. Without completing this exercise in full detail, note that because field reads and writes do not obey quite the same restrictions, handling reads effectively doubles the number of row variables and bounds (for every method signature), though the bounds for reading are slightly more relaxed than those for writing (since fields may be local *or* inherited). Reading from an inherited field is acceptable and is in fact how method dispatch commonly works in JavaScript. With the code in Figure 3, calling `f.incAndCount()` should be permitted, even though the body of that method, inherited from the prototype, invokes (and therefore reads) two inherited method fields. Extending for method-read sets results in types like this one, which adds more complex constraints to deal with the fact that reading writable fields is safe:

$$\exists R, W :: \text{row}. \exists X \subseteq W \exists Y \subseteq (R \cup W). \{ \text{inc} : () \xrightarrow{\text{wr}(X)} (), \text{get} : () \xrightarrow{\text{rd}(Y)} \text{number}, R \mid W \}$$

Second, we have not addressed the additional complication mentioned earlier: the receiver may escape a method, so tracking *only* the receiver fields a method modifies is insufficient! Consider a method body that registers the receiver for updates:

```
Foo.prototype.reregister = function() {  
  this.targetSource.registerListener(this);  
}
```

If the `registerListener` method modifies its argument (directly or by invoking methods that do so), those modifications should also be reflected in the effect of the `reregister` method. But the only way for this to work is if the type or effect of `registerListener` reflects the fields it updates *on its arguments*, as well as on its receiver (`this.targetSource` in this case). This also brings in the aliasing issues discussed in the effect system reconstruction of ReIm.

As presented here, the complexity is clearly significant even before it is carried to its logical conclusion. But at what point during this design process did it *become too complex*? Can we identify a point in this design evolution where it clearly crossed the line? The project

required structural types for objects from the start, so it's hard to tell exactly which pieces of the growth above are truly necessary and which add too much complexity: rows for instance originated in type inference for record calculi [64], and these kinds of constraints between rows were known to be necessary to type certain kinds of programs [4]. The project goals included regular developers using the result, so inference was a requirement, which then implied rows and row constraints had a role to play. The eventual implementation uses rows, though row *constraints* are limited to type inference only and ultimately do not appear in surface types seen by developers. Many type systems with unpleasant core complexity manage to tame some of it through convenient short-hands and careful selection of default assumptions. So while hindsight shows this approach would have led to more complex metatheory and implementation, and probably significantly worse error reporting, the fact that this approach had some justification in its relationship to inference, and clearly exposed all of the required information, made it harder to tell when this route might have crossed the line to being unacceptably complex.

A fair question to ask at this point is also how much of this complexity stems from the particular problem at hand – reasoning about the particular interaction of field reads and writes with JavaScript's uncommon inheritance model. Greenhouse and Boyland's work on an object-oriented analogue [30] of FX [40, 24] (an effect system for reasoning about non-interference of program expressions) resembles early stages of the development outlined here. They continued the FX emphasis on regions, and permitted Java classes to declare abstract regions of fields. Regions existed in a nesting hierarchy (which inspired the same structure in DPJ [1]), such as a hashtable having nested regions for keys and values to separate impacts on those parts of the structure. Method effects were then the set of regions read or written by the method, with field names acting as special (very specific) regions. Effects could refer to specific object (e.g., the value region of a hashtable taken as a parameter), which is roughly analagous to the outline we gave for handling the `reregister` example. As a result actually checking their effect system requires points-to information [30, 2].

3.3.2 Back to Capabilities for Invariants

Starting from the outline above, how did we simplify the system? We can see several steps to condense the information from our hypothetical complex effect system down to the still-sophisticated, but more manageable published system [7]. The first step was to simply impose a single upper bound on the written receiver fields, shared across all methods on that object. Thus, object types would (sometimes) contain *two* kinds of object types: a physical type describing the local and inherited fields (which fields are actually present, and which are writable), and a method-required type describing sufficient receiver assumptions to execute any attached methods. This moves part of the effect information from the methods to the object type itself (and is a feature of the final system). The published system calls the method-required portion of the type the *method-accessed fields*. Because both present and method-access fields must further split into distinctions between possibly-inherited (and therefore readable) and definitely-local (and therefore writable), this resulted at one point in four-part object types

$$\left\{ \begin{array}{c} \text{Physically present fields} \\ \overbrace{\left\{ \begin{array}{c} r \quad | \quad w \\ \text{Readable} \quad \text{Writable} \end{array} \right\}} \\ \text{Method-accessed fields} \\ \overbrace{\left\{ \begin{array}{c} mr \quad | \quad mw \\ \text{Method-read} \quad \text{Method-written} \end{array} \right\}} \end{array} \right\}$$

where each variable is a row:

10:20 Designing with Static Capabilities and Effects: Use, Mention, and Invariants

- r contains definitely-present, but possibly-inherited fields, which are safe to read.
- w contains definitely-present, definitely-local fields, which are safe to write.
- mr contains fields that may be read by some method, but are definitely not written by any method of the object.
- mw contains fields that may be written by some method of the object.

The method-access fields are taken to be a single upper bound on the effect of any method on the object, dualized to describe the *capabilities sufficient to execute any method on the object*. The other fields describe the physically present fields of the object, distinguishing those that are definitely local and can therefore be written without affecting object layout.

Then using the read-write split on physical fields (r and w) then becomes apparent as a way to summarize how a method uses its arguments – if `registerListener` above modifies field `foo` of its argument, it will be reflected in the required parameter type containing a writable field `foo` of the appropriate type, which we can interpret as a reference capability required by `registerListener`. Since the types in the system already needed to track which fields are on the immediate referent (and therefore, safe to write without changing field layout) and which are possibly-inherited (so safe to read, but not necessarily safe to write), this actually removes some redundancy: the physical layout information plays double-duty as both a physical description *and* a capability granting read-access to present fields and write-access to local fields. And while it again begins to sacrifice the use-mention distinction, for this problem the distinction turns out not to be critical.

“Flattening” use information from effects into mention information in reference types (capabilities) addresses the issue of soundly tracking reads and writes. This leaves us with two other challenges raised above: reasoning about when it is actually safe to invoke a method, and abstracting types in a way that we can invoke methods based on interfaces with different implementations. Turning to the notion of asymmetric compatible capabilities that collaboratively enforce an invariant, we find another solution. When deciding whether it is safe to invoke a method, it is not really relevant which particular fields are present, only that those present include the ones accessed by methods (again, informally blurring some distinctions between reads and writes).

We can shift our view to maintaining each object as being either an *abstract* object (whose methods access fields that are not present, by analogy to an abstract class), or a *concrete* object with all the fields required (in the appropriate places) to safely invoke *any* of its methods (since there is now only one common bound on the behavior of all methods on an object). We can view membership in one of these sets as an invariant collaboratively maintained by all references to an object. Given one of the “double” object types suggested above, the check is simple: if every field assumed writable or readable by methods (i.e., in the method-accessed fields) is actually writable or readable on the physical object (i.e., in the right partition of the physically-present fields), then it is safe to invoke methods on that object. Moreover, once that check is performed for a given object, since the method-access field information for the object and the physical layout information should be invariant, the information about method-accessed fields can be discarded, leaving only the basic physical object type (r and w) as important.

For example, consider `f` and `g` from our earlier example. `f` would be given (full) type:

$$\{inc : () \rightarrow () \mid x : \text{number} \mid \emptyset \mid x : \text{number}\}$$

and `g` would receive the analogous type mentioning y :

$$\{inc : () \rightarrow () \mid y : \text{number} \mid \emptyset \mid y : \text{number}\}$$

Since the method-written set is contained in the physically local writable set for each object, f can be given the simpler object type $\{inc : () \rightarrow () \mid x : \text{number}\}^{\text{NC}}$, where NC tags the object as *concrete*, indicating the check was performed when method-accessed fields were known, and aliases will ensure that check remains true. g can be given the analogous type mentioning y , and then traditional width subtyping¹² lets both be given the common supertype $\{inc : () \rightarrow () \mid \emptyset\}^{\text{NC}}$. This common supertype only mentions the method of interest, using standard subtyping to hide the irrelevant differences. But because it is flagged as concrete, the type system can permit the increment method to be invoked: the NC tag indicates the referent already satisfies sufficient invariants for any method invocation to be safe, and restrictions on how aliases are created (essentially, sound treatment of subtyping and field updates) ensure the invariant is preserved. Most people would agree this is substantially simpler than the type laden with explicit row quantification and constraints.

The only time the full double object types are required is when handling prototype objects (e.g., for initialization) or replacing existing methods. In those cases, it is necessary to check that the method-required half of the object type is (informally) a subtype of the *assumed receiver type* of a newly-installed method. Intuitively, that method-accessed sets are an invariant of the object, and attaching a method ensures the new method preserves that invariant (i.e., does not install a method that accesses other things). Read as capabilities, the full object types provide the extra information / permissions required to check method replacement, which takes the form of unattached methods with assumed receiver types stating the permissions required by the new method body. Chandra et al. call these full types *prototypal types*, and distinguish them from *non-prototypal types* that carry no method-accessed fields because they can only be created from prototypal types when the check that all method-accessed fields are present succeeds. In some cases complete objects may also be used as prototypes, so some objects may be aliased by references with dual types (prototypal) and by references with single types (non-prototypal). The non-prototypal concrete (i.e., NC) types grant the capability to invoke any visible methods. The dual (prototypal) types grant the capabilities to modify prototype or method members (and carry sufficient information to actually perform the containment checks between local fields and method assumptions).

While the discussion above focused on reasoning about access to specific fields, it is worth noting that all structural object types – including those just discussed – form a sort of reference capability with support for static delegation (but not revocation). If a developer wishes to pass an object to some code, but limit which methods of the object may be invoked, using width subtyping one can obtain a reference which does not mention the “restricted” operations, and a sound type system (and limiting reflection) ensures a callee will not

4 Conclusion

We have outlined what we have found to be the major trade-offs in practice between static (reference) capabilities and effect systems: choosing between simpler design and abstract reasoning principles, and handling the use-mention distinction. We have also highlighted examples of a subtle interplay between reference capabilities and modest aspects of type systems (weakening rules and type contexts) that results in useful added expressive power in a way that has not been highlighted previously. Lastly, we have tried to put these in context by explaining what breaks – functionally, or by introducing unwieldy complexity – when considering effect system versions of reference capability systems or vice versa, based on our personal experience facing these trade-offs while designing reference capability systems and

¹²Tweaked for the read/write split of fields.

effect systems. We hope primarily that this will be useful to others in choosing between approaches to static reasoning, and helpful to newcomers seeking to better understand the trade-offs between these approaches.

References

- 1 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.
- 2 John Boyland and Aaron Greenhouse. Mayequal: A new alias question. In *International Workshop on Aliasing in Object-Oriented Systems (IWAOOS)*, 1999.
- 3 John Boyland, James Noble, and William Retert. Capabilities for sharing. In *European Conference on Object-Oriented Programming*, pages 2–27. Springer, 2001.
- 4 Luca Cardelli and John C Mitchell. Operations on records. *Mathematical structures in computer science*, 1(01):3–48, 1991.
- 5 Elias Castegren and Tobias Wrigstad. Reference capabilities for concurrency control. In *30th European Conference on Object-Oriented Programming, ECOOP 2016*, 2016.
- 6 Elias Castegren and Tobias Wrigstad. Relaxed linear references for lock-free programming. In *31st European Conference on Object-Oriented Programming, ECOOP 2017*, 2017.
- 7 Satish Chandra, Colin S. Gordon, Jean-Baptiste Jeannin, Cole Schlesinger, Manu Sridharan, Frank Tip, and Young-Il Choi. Type Inference for Static Compilation of JavaScript. In *Proceedings of the 2016 ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2016)*, Amsterdam, The Netherlands, November 2016. doi:10.1145/2983990.2984017.
- 8 Jeffrey S. Chase, Henry M. Levy, Edward D. Lazowska, and Miche Baker-Harvey. Lightweight shared objects in a 64-bit operating system. In *Conference Proceedings on Object-oriented Programming Systems, Languages, and Applications, OOPSLA '92*, pages 397–413, New York, NY, USA, 1992. ACM. doi:10.1145/141936.141969.
- 9 Philip Wontae Choi, Satish Chandra, George Necula, and Koushik Sen. SJS: a Typed Subset of JavaScript with Fixed Object Layout. Technical Report UCB/EECS-2015-13, UC Berkeley, 2015.
- 10 Dave Clarke, Johan Östlund, Ilya Sergey, and Tobias Wrigstad. Ownership types: A survey. In *Aliasing in Object-Oriented Programming. Types, Analysis and Verification*, pages 15–58. Springer, 2013.
- 11 David G Clarke, John M Potter, and James Noble. Ownership types for flexible alias protection. In *Proceedings of the 13th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 48–64, 1998.
- 12 Sylvan Clebsch, Sophia Drossopoulou, Sebastian Blessing, and Andy McNeil. Deny capabilities for safe, fast actors. In *Proceedings of the 5th International Workshop on Programming Based on Actors, Agents, and Decentralized Control*, pages 1–12. ACM, 2015.
- 13 Aaron Craig, Alex Potanin, Lindsay Groves, and Jonathan Aldrich. Capabilities: Effects for Free. In *International Conference on Formal Engineering Methods (ICFEM)*, 2018.
- 14 Donald Davidson. Quotation. *Theory and decision*, 11(1):27, 1979.
- 15 David L. Detlefs, K. Rustan M. Leino, and Greg Nelson. Wrestling with rep exposure. Technical Report SRC-RR-156, Digital Equipment Corporation, July 1998. URL: <https://www.hp1.hp.com/techreports/Compaq-DEC/SRC-RR-156.html>.
- 16 Werner Dietl, Sophia Drossopoulou, and Peter Müller. Generic Universe Types. In E. Ernst, editor, *ECOOP*, volume 4609 of *Lecture Notes in Computer Science*, pages 28–53, Berlin, Germany, July 2007. Springer-Verlag.
- 17 Werner Dietl and Peter Müller. Universes: Lightweight ownership for jml. *Journal of Object Technology*, 4(8):5–32, 2005.

- 18 Sophia Drossopoulou, James Noble, Mark S Miller, and Toby Murray. Permission and authority revisited towards a formalisation. In *Proceedings of the 18th Workshop on Formal Techniques for Java-like Programs*, pages 1–6, 2016.
- 19 Joe Duffy. Blogging about Midori, November 2015. URL: <http://joeduffyblog.com/2015/11/03/blogging-about-midori/>.
- 20 Andrzej Filinski. Monads in action. In *POPL*, 2010.
- 21 Jeffrey S. Foster, Manuel Fähndrich, and Alexander Aiken. A theory of type qualifiers. In *Proceedings of the ACM SIGPLAN 1999 Conference on Programming Language Design and Implementation*, PLDI '99, pages 192–203. ACM, 1999. doi:10.1145/301618.301665.
- 22 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.
- 23 Paola Giannini, Marco Servetto, Elena Zucca, and James Cone. Flexible recovery of uniqueness and immutability. *Theoretical Computer Science*, 764:145–172, 2019.
- 24 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.
- 25 Colin S. Gordon. *Verifying Concurrent Programs by Controlling Alias Interference*. PhD thesis, University of Washington, Seattle, WA, USA, August 2014. URL: <https://digital.lib.washington.edu/researchworks/handle/1773/26020>. URL: <https://digital.lib.washington.edu/researchworks/handle/1773/26020>.
- 26 Colin S. Gordon, Werner Dietl, Michael D. Ernst, and Dan Grossman. JavaUI: Effects for Controlling UI Object Access. In *ECOOP*, 2013.
- 27 Colin S. Gordon, Michael D. Ernst, and Dan Grossman. Rely-Guarantee References for Refinement Types Over Aliased Mutable Data. In *PLDI*, Seattle, WA, USA, June 2013. doi:10.1145/2491956.2462160.
- 28 Colin S. Gordon, Michael D. Ernst, Dan Grossman, and Matthew J. Parkinson. Verifying Invariants of Lock-free Data Structures with Rely-Guarantee and Refinement Types. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 39(3), May 2017. doi:10.1145/3064850.
- 29 Colin S. Gordon, Matthew J. Parkinson, Jared Parsons, Aleks Bromfield, and Joe Duffy. Uniqueness and Reference Immutability for Safe Parallelism. In *OOPSLA*, Tucson, AZ, USA, October 2012. doi:10.1145/2384616.2384619.
- 30 Aaron Greenhouse and John Boyland. An object-oriented effects system. In *European Conference on Object-Oriented Programming*, pages 205–229. Springer, 1999.
- 31 Philipp Haller and Martin Odersky. Capabilities for Uniqueness and Borrowing. In *ECOOP*, 2010.
- 32 Wei Huang, Werner Dietl, Ana Milanova, and Michael D. Ernst. Inference and checking of object ownership. In *European Conference on Object-Oriented Programming (ECOOP 2012)*, pages 181–206. Springer, 2012.
- 33 Wei Huang, Ana Milanova, Werner Dietl, and Michael D. Ernst. Reim & reiminfer: Checking and inference of reference immutability and method purity. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '12, pages 879–896. ACM, 2012.
- 34 Butler W Lampson. Protection. *ACM SIGOPS Operating Systems Review*, 8(1):18–24, 1974.
- 35 Henry M Levy. *Capability-based computer systems*. Digital Press, 1984. URL: <https://homes.cs.washington.edu/~levy/capabook/>.
- 36 Paul Liétar. Formalizing Generics for Pony, 2017. Imperial College London Bachelor's Thesis.
- 37 Sam Lindley and James Cheney. Row-based effect types for database integration. In *Proceedings of the 8th ACM SIGPLAN workshop on Types in language design and implementation*, pages 91–102. ACM, 2012.

- 38 Fengun Liu, Sandro Stucki, Nada Amin, Paolo Giosuè, and Martin Odersky. Stoic: Towards Disciplined Capabilities. Technical report, École Polytechnique Fédérale de Lausanne, 2020. URL: <https://infoscience.epfl.ch/record/273642>.
- 39 Fengyun Liu. A Study of Capability-Based Effect Systems, 2016. Master of Computer Science Thesis, École Polytechnique Fédérale de Lausanne. URL: <https://infoscience.epfl.ch/record/219173>.
- 40 J. M. Lucassen and D. K. Gifford. Polymorphic Effect Systems. In *POPL*, 1988.
- 41 Daniel Marino and Todd Millstein. A Generic Type-and-Effect System. In *TLDI*, 2009. doi:10.1145/1481861.1481868.
- 42 Darya Melicher, Yangqingwei Shi, Valerie Zhao, Alex Potanin, and Jonathan Aldrich. Using Object Capabilities and Effects to Build and Authority-Safe Module System. In *Workshop on Object-Capability Languages, Systems, and Applications (OCAP)*, 2017.
- 43 Filipe Militão, Jonathan Aldrich, and Luís Caires. Rely-Guarantee Protocols. In *28th European Conference on Object-Oriented Programming, ECOOP 2014*, 2014.
- 44 Filipe Militão, Jonathan Aldrich, and Luís Caires. Composing interfering abstract protocols. In *30th European Conference on Object-Oriented Programming, ECOOP 2016*, 2016. doi:10.4230/LIPIcs.ECOOP.2016.16.
- 45 Mark Samuel Miller. *Robust Composition: Towards a Unified Approach to Access Control and Concurrency Control*. PhD thesis, Johns Hopkins University, Baltimore, Maryland, USA, May 2006.
- 46 Eugenio Moggi. Computational lambda-calculus and monads. In *LICS*, 1989.
- 47 AW Moore. How significant is the use/mention distinction? *Analysis*, 46(4):173–179, 1986.
- 48 Karl Naden, Robert Bocchino, Jonathan Aldrich, and Kevin Bierhoff. A type system for borrowing permissions. In *Proceedings of the 39th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 557–570, 2012.
- 49 James Noble, Jan Vitek, and John Potter. Flexible alias protection. In *European Conference on Object-Oriented Programming*, pages 158–185. Springer, 1998.
- 50 Leo Oswald, Grégory Essertel, Xilun Wu, Lilliam I González Alayón, and Tiark Rompf. Gentrification gone too far? affordable 2nd-class values for fun and (co-) effect. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 234–251, 2016.
- 51 Alex Potanin, Monique Damitio, and James Noble. Are your incoming aliases really necessary? counting the cost of object ownership. In *2013 35th International Conference on Software Engineering (ICSE)*, pages 742–751. IEEE, 2013.
- 52 François Pottier. Hiding local state in direct style: a higher-order anti-frame rule. In *2008 23rd Annual IEEE Symposium on Logic in Computer Science*, pages 331–340. IEEE, 2008.
- 53 John C Reynolds. Separation logic: A logic for shared mutable data structures. In *Proceedings 17th Annual IEEE Symposium on Logic in Computer Science*, pages 55–74. IEEE, 2002.
- 54 Lukas Rytz, Nada Amin, and Martin Odersky. A flow-insensitive, modular effect system for purity. In *Proceedings of the 15th Workshop on Formal Techniques for Java-like Programs*, page 4. ACM, 2013.
- 55 Benno Stein, Lazaro Clapp, Manu Sridharan, and Bor-Yuh Evan Chang. Safe stream-based programming with refinement types. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, ASE 2018*, pages 565–576, New York, NY, USA, 2018. ACM. doi:10.1145/3238147.3238174.
- 56 Jean-Pierre Talpin and Pierre Jouvelot. Polymorphic type, region and effect inference. *Journal of functional programming*, 2(03):245–271, 1992. doi:10.1017/S095679680000393.
- 57 Mads Tofte and Jean-Pierre Talpin. Implementation of the Typed Call-by-value λ -calculus Using a Stack of Regions. In *POPL*, 1994.
- 58 Mads Tofte and Jean-Pierre Talpin. Region-based memory management. *Information and computation*, 132(2):109–176, 1997.

- 59 Matthew S. Tschantz and Michael D. Ernst. Javari: Adding Reference Immutability to Java. In *OOPSLA*, 2005. doi:10.1145/1094811.1094828.
- 60 Tom Van Cutsem and Mark S. Miller. Proxies: Design principles for robust object-oriented intercession apis. In *Proceedings of the 6th Symposium on Dynamic Languages, DLS '10*, pages 59–72, New York, NY, USA, 2010. ACM. doi:10.1145/1869631.1869638.
- 61 Tom Van Cutsem and Mark S Miller. Trustworthy proxies. In *European Conference on Object-Oriented Programming*, pages 154–178. Springer, 2013.
- 62 David Walker. Substructural type systems. In *Advanced topics in types and programming languages*, pages 3–44. The MIT Press, 2005.
- 63 David Walker, Karl Crary, and Greg Morrisett. Typed memory management via static capabilities. *ACM Trans. Program. Lang. Syst.*, 22(4):701–771, July 2000. doi:10.1145/363911.363923.
- 64 Mitchell Wand. Type inference for record concatenation and multiple inheritance. In *Logic in Computer Science, 1989. LICS'89, Proceedings., Fourth Annual Symposium on*, pages 92–97. IEEE, 1989.
- 65 Yoav Zibin, Alex Potanin, Mahmood Ali, Shay Artzi, Adam Kiezun, and Michael D. Ernst. Object and Reference Immutability Using Java Generics. In *ESEC-FSE*, 2007. doi:10.1145/1287624.1287637.
- 66 Yoav Zibin, Alex Potanin, Paley Li, Mahmood Ali, and Michael D. Ernst. Ownership and Immutability in Generic Java. In *OOPSLA*, 2010. doi:10.1145/1869459.1869509.

Owicki-Gries Reasoning for C11 RAR

Sadegh Dalvandi 

University of Surrey, United Kingdom
m.dalvandi@surrey.ac.uk

Simon Doherty 

University of Sheffield, United Kingdom
s.doherty@sheffield.ac.uk

Brijesh Dongol 

University of Surrey, United Kingdom
b.dongol@surrey.ac.uk

Heike Wehrheim 

Paderborn University, Germany
wehrheim@upb.de

Abstract

Owicki-Gries reasoning for concurrent programs uses Hoare logic together with an *interference freedom* rule for concurrency. In this paper, we develop a new proof calculus for the C11 RAR memory model (a fragment of C11 with both relaxed and release-acquire accesses) that allows all Owicki-Gries proof rules for compound statements, including non-interference, to remain unchanged. Our proof method features novel assertions specifying *thread-specific views* on the state of programs. This is combined with a set of Hoare logic rules that describe how these assertions are affected by atomic program steps. We demonstrate the utility of our proof calculus by verifying a number of standard C11 litmus tests and Peterson’s algorithm adapted for C11. Our proof calculus and its application to program verification have been fully mechanised in the theorem prover Isabelle.

2012 ACM Subject Classification Theory of computation → Logic and verification; Theory of computation → Hoare logic; Theory of computation → Concurrency; Theory of computation → Operational semantics; Theory of computation → Program reasoning

Keywords and phrases C11, Verification, Hoare logic, Owicki-Gries, Isabelle

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2020.11

Supplementary Material ECOOP 2020 Artifact Evaluation approved artifact available at <https://doi.org/10.4230/DARTS.6.2.15>.

Funding *Sadegh Dalvandi*: Supported by EPSRC Grant EP/R032556/1.

Simon Doherty: Supported by EPSRC Grant EP/R032351/1.

Brijesh Dongol: Supported by EPSRC Grant EP/R032556/1.

Heike Wehrheim: Supported by DFG grant WE 2290/12-1.

1 Introduction

In 1976, Susan Owicki and David Gries proposed an extension of Hoare’s axiomatic reasoning technique [15] to concurrent programs [25]. Their proof calculus allows one to reason about concurrent programs with shared variables via a number of proof rules, including the rules for sequential programs as introduced by Hoare plus an additional proof rule for concurrent composition. This composition rule basically allows for the conjunction of pre- and post-conditions of the process’ individual proofs, given that their proof outlines are *interference free*. Interference freedom requires that an assertion in the proof of one process cannot be invalidated by a statement in another process, when executed under the statement’s precondition.



© Sadegh Dalvandi, Simon Doherty, Brijesh Dongol, and Heike Wehrheim;
licensed under Creative Commons License CC-BY

34th European Conference on Object-Oriented Programming (ECOOP 2020).

Editors: Robert Hirschfeld and Tobias Pape; Article No. 11; pp. 11:1–11:26

Leibniz International Proceedings in Informatics



Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany



Today, concurrent programs are run on multi-core processors. Multi-core processors come with *weak memory models* specifying the execution behaviour of concurrent programs. Reasoning consequently needs to be adapted to the memory model under consideration. Owicki-Gries reasoning is, however, fixed to the memory model of *sequential consistency* (SC) [23], and is unsound for weak memory models. Recent research has thus worked towards new sound proof calculi for concurrent programs. Most often, such approaches involve concurrent separation logics (e.g., GPS and RSL [32, 16]). These techniques constitute a radical departure from the (relatively) small and easy proof calculus of Owicki and Gries, further extending already complex logics. A proposal for a (rely-guarantee variant of) the Owicki-Gries proof system has been made by Lahav and Vafeiadis [21], however, requiring a strengthened non-interference check.

In this paper, we develop a proof method based on the Owicki-Gries proof calculus, keeping all of the original proof rules including the non-interference check unchanged. Our technique introduces a set of basic axioms to cope with memory accesses (reads, writes, read-modify-writes) and simple assertions that describe the current configuration of the weak memory state. Our proof calculus targets the weak memory model of the C11 programming language [8]. Here, we deal with the release-acquire-relaxed (RAR) fragment of C11 (thereby going further than prior work on Owicki-Gries reasoning for C11 [21]).

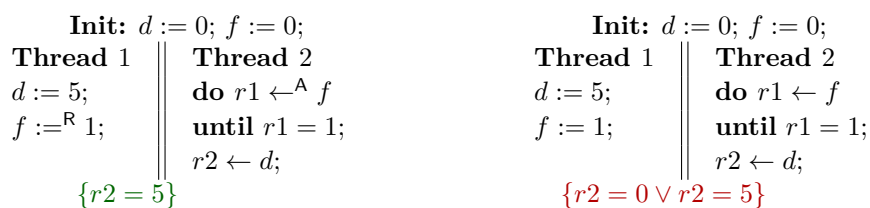
The key idea of our approach is the usage of novel assertions which enables the specification of *thread-specific* views on shared variables. We also include a specific assertion containing a modality for release-acquire (RA) synchronisation, capturing particularities of C11 RA message passing. The use of non-standard assertions as a consequence necessitates the introduction of new rules of assignment, formalising the effect of assignments on assertions.

We build our proof calculus on top of an *operational* semantics for C11 RAR. The semantics is a mixture of the operational semantics proposed by Doherty et al. [12] (for RAR) and Kaiser et al.'s semantics [16] for RA plus non-atomics. Correctness of this novel proposal is shown by proving it to coincide with the semantics defined by Doherty et al. [12] which in turn has been proven to coincide with the standard axiomatic semantics of Batty et al. [8]. We have formalised our semantics within the theorem prover Isabelle [26] and mechanically proved soundness of all of our new rules for C11 assertions. Moreover, we provide mechanical proofs of several litmus tests from the literature (message passing, load buffering, read-read coherence) as well as a version of Peterson's algorithm adapted for C11 memory [12, 34].

Overview. The paper is organised as follows. In the next section we start with an example explaining the behaviour of concurrent programs on C11, motivating our novel assertions. Section 3 defines the syntax of C11 RAR programs and Section 4 its semantics. We present the proof calculus and its novel assertions in Section 5 via proofs of correctness for some standard litmus tests, and a case study of Peterson's algorithm in Section 6. Section 7 describes our Isabelle mechanisation, Section 8 discusses related work and the last section concludes.

2 Deductive Reasoning for Weak Memory

In this section, we illustrate the basic principles of C11 synchronisation and our verification method by considering the message-passing example (Figures 1 and 2). The two programs are almost identical and consist of two threads executing in parallel, accessing shared variables. The assertions in curly brackets at the end specify the programs' postconditions.



■ **Figure 1** Message-passing litmus test.

■ **Figure 2** Unsynchronised message passing.

The programs comprise two shared variables: d (that stores some data) and f (that stores a flag). In both programs, both d and f are initially 0. thread 1 updates d to 5, then updates f to 1. Thread 2 waits for f to be set to 1, then reads from d . Under sequential consistency, one would expect that the final value of $r2$ is 5, since the loop in thread 2 only terminates after f has been updated to 1 in thread 1, which in turn happens after d has been set to 5. However, the C11 semantics allows the behaviour in Figure 2, where thread 2 may read a stale value of d , and hence only the weaker postcondition $r2 = 0 \vee r2 = 5$ holds. To regain the expected behaviour, one must introduce additional synchronisation in the program. In particular, the write to f by thread 1 must be a *releasing write* (i.e., $f :=^R 1$) and the read of f in thread 2 must be an *acquiring read* (i.e., $r1 \leftarrow^A f$) as in Figure 1.

In sequential consistency all threads have a single common view of the shared state, namely all threads see the latest write that occurs for each variable. When a new write is executed, the views of all threads are updated so that they see this write. In contrast, each thread in C11 programs has its own view of each variable, which is affected by synchronisation annotations. Thus, for the program in Figure 2, after initialisation, all threads see the initial writes (i.e., $d = 0, f = 0$). The assignments in thread 1 only change thread 1's view, and leave thread 2's view unchanged. Thus, after execution of $f := 1$, thread 2 has access to two values for d (i.e., $d \in \{0, 5\}$) and f (i.e., $f \in \{0, 1\}$). Even if thread 2 reads $f = 1$, its view of d remains unchanged and it continues to have access to both values of d .

The program in Figure 1 has a similar semantics for initialisation and execution of thread 1, i.e., its execution does not affect the view of thread 2. However, due to the release-acquire synchronisation on f (notation R and A), after thread 2 reads $f = 1$, its view for d will be updated so that the stale value $d = 0$ is no longer available for it to read. One way to explain this behaviour is by thinking of thread 1 as *passing its knowledge of the write to d* to thread 2 via the variable f , which is synchronised using the release-acquire annotations.

This intuition is captured formally using a semantics based on *timestamps* [16, 13, 17, 27], which enables one to encode each thread's view and define how these views are updated. In this paper, we characterise the release-acquire-relaxed subset of C11 [12] (C11 RAR) using timestamps, which has a restriction prohibiting the so-called *load-buffering* litmus test¹ [22].

The main contribution of our paper is an assertion language that enables one to reason about thread views in a Hoare-style proof calculus, resulting in the proof outline given in Figure 3. As already noted, the key advantage of these assertions is the fact that standard rules of Hoare and Owicki-Gries logic remain unchanged. For message passing, we require three main types of assertions (see Section 5):

Possible value. A possible value assertion (denoted $x \approx_t n$) states that thread t can read value n of global variable x , i.e., there is a write to x with value n beyond or including the *viewfront*² of thread t . Note that there may be more than one such write, and hence

¹ Litmus tests are small code snippets with particularly interesting behaviour.

² We borrow the term viewfront from Popkadaev et al. [27].

$$\begin{array}{c}
\text{Init: } d := 0; f := 0; \\
\{f =_1 0 \wedge f =_2 0 \wedge d =_1 0 \wedge d =_2 0\} \\
\text{Thread 1} \quad \parallel \quad \text{Thread 2} \\
\{f \not\approx_2 1 \wedge d =_1 0\} \quad \parallel \quad \{[f = 1](d =_2 5)\} \\
1 : d := 5; \quad \parallel \quad 3 : \text{do } r1 \leftarrow^A f \text{ until } r1 = 1; \\
\{f \not\approx_2 1 \wedge d =_1 5\} \quad \parallel \quad \{d =_2 5\} \\
2 : f :=^R 1; \quad \parallel \quad 4 : r2 \leftarrow d; \\
\{true\} \quad \parallel \quad \{r2 = 5\} \\
\{r2 = 5\}
\end{array}$$

■ **Figure 3** Proof outline for message passing.

there may be several possible values for a given variable. For instance, there might be one write to x with value v_1 in thread t 's viewfront and two more writes to x with values v_2 and v_3 beyond the viewfront. Then assertions $x \approx_t v_1$, $x \approx_t v_2$ and $x \approx_t v_3$ all hold.

Definite value. A definite value assertion (denoted $x =_t n$) states that thread t 's viewfront is up-to-date with the writes to x (i.e., there is a single write to x beyond or including the viewfront of thread t), and this write updates x 's value to n . Thus, t definitely knows the variable x to have value n .

Conditional value. A conditional value assertion (denoted $[x = n](y =_t m)$) captures the message passing idiom for variable y via variable x . It guarantees that when thread t reads x to be n via an acquiring read, a release-acquire synchronisation is induced and thereby t learns the definite value of y to be m . In particular, after reading $x = n$ via an acquiring read, the viewfront for t is updated so that the only write to y beyond or including this viewfront is a write with value m .

For the example in Figure 3, after initialisation, both threads 1 and 2 have definite value 0 for both d and f . The precondition of $d := 5$ states that thread 2 cannot possibly observe 1 for f (i.e., $f \not\approx_2 1$, needed for interference freedom of proof outlines) and thread 1 definitely observes 0 for d (i.e., $d =_1 0$). These assertions can be proven *locally correct* and *interference free* since thread 2 neither modifies d nor f . The precondition of $f :=^R 1$ is similar but with $d =_1 5$ in place of $d =_1 0$. The precondition of the **until** loop in thread 2 contains a conditional value assertion, which ensures that if thread 2 reads $f = 1$ then it will definitely read $d = 5$. This conditional value assertion enables one to establish local correctness of the precondition (i.e., $d =_2 5$) of the statement $r2 \leftarrow d$, which leads to the postcondition of the program. Each of the assertions in thread 2 can be proven to be interference free against thread 1.

3 Program Syntax

We start by defining the syntax of concurrent programs, starting with the structure of sequential programs (single threads). A thread may use *global* shared variables (from Var_G) and local registers (from Var_L). We let $Var = Var_G \cup Var_L$ and assume $Var_G \cap Var_L = \emptyset$. Global variables can be accessed in three different *synchronisation modes*: acquire (A, for reads), release (R, for writes) and relaxed (no annotation). The annotation RA is employed for *update* operations, which read and write to a shared variable in a single atomic step. We use x, y, z to range over global variables and $r1, r2, \dots$ to range over local variables. We assume that \ominus is a unary operator (e.g., \neg), \oplus is a binary operator (e.g., $\wedge, +, =$) and n

is a value (of type Val). Expressions may only involve local variables. For a treatment of expressions with global variables in the semantics see [12]. The syntax of sequential programs, Com , is given by the following grammar (with $r \in Var_L, x \in Var_G$):

$$\begin{aligned} Exp_L &::= Val \mid r \mid \ominus Exp_L \mid Exp_L \oplus Exp_L \\ ACom &::= \mathbf{skip} \mid x.\mathbf{swap}(n)^{RA} \mid r := Exp_L \mid x :=^{[R]} Exp_L \mid r \leftarrow^{[A]} x \\ Com &::= ACom \mid Com; Com \mid \mathbf{if} B \mathbf{then} Com \mathbf{else} Com \mid \mathbf{while} B \mathbf{do} Com \end{aligned}$$

where we assume B to be an expression of type Exp_L that evaluates to a boolean. The statement $x.\mathbf{swap}(n)^{RA}$ atomically reads the variable x (using an acquiring read) and updates x to value n (using a releasing write) in a single atomic step. Its execution therefore gives rise to an atomic read-modify-write update event. We have not included a **CAS** operation here; it could similarly be implemented by an update event (see e.g. [33]).

The notation $[X]$ denotes that the annotation X is optional, where $X \in \{A, R\}$, enabling one to distinguish relaxed, acquiring and releasing accesses. Loops will be used in other forms, like **do-until** or **do-while**, which are straightforward to define in terms of the command syntax above.

As is standard in Owicki-Gries proofs, we make use of *auxiliary variables*, which are variables that do not affect the meaning of a program, but appear in proof assertions. We require that each auxiliary variable is *local* to the thread in which it occurs. Auxiliary variables may only occur in assignments, not in conditional statements, and only in the form $\alpha := E$, where $E \in Exp_L$ and α is an auxiliary variable³. Finally, we require that writes to auxiliary variables occur atomically in conjunction with another (non-auxiliary) atomic program step. Such atomic operations are written as $\langle A, \alpha := E \rangle$, where $A \in ACom$. This is more of a technical requirement which could also easily be relaxed. It guarantees that the programs without and with auxiliary variables have the same number of transitions (no stuttering steps).

For simplicity, we assume concurrency at the top level only. We let Tid be the set of all thread identifiers and use a function $Prog : Tid \rightarrow Com$ to model a program comprising multiple threads. In examples, we typically write concurrent programs as $C_1 \parallel \dots \parallel C_n$, where $C_i \in Com$. We further assume some initialisation of variables. The structure of our programs thus is **Init**; $(C_1 \parallel \dots \parallel C_n)$.

4 Semantics

The operational semantics for this language is defined in two parts. The *program semantics* fixes the steps that the concurrent program can take. This gives rise to transitions $(P, lst) \xrightarrow{a}_t (P', lst')$ of a thread t where P and P' are programs, lst and lst' is the state of local variables and a is an action (possibly the silent action τ , see below). The program semantics is combined with a *memory semantics* which reflects the C11 state (denoted by σ), and in particular the write actions from which a read action can read.

We start by fixing the actions, where $x \in Var_G$ and $m, n \in Val$:

$$\mathbf{Act} = \{rd(x, n), rd^A(x, n), wr(x, n), wr^R(x, n), upd^{RA}(x, n, m)\}$$

containing actions for (releasing) reads, (acquiring) writes and updates (reading value n and writing m). We furthermore employ a silent τ action and let $\mathbf{Act}_\tau = \mathbf{Act} \cup \{\tau\}$. For an action $a \in \mathbf{Act}$, we let $var(a) \in Var_G$ be the variable read (or written to), $rdval(a) \in Val$ be the

³ The locality requirement is the only difference to “normal” Owicki-Gries auxiliary variables.

$$\begin{array}{c}
 \frac{r \in \text{Var}_L \quad n = \llbracket E \rrbracket_{ls}}{(r := E, ls) \xrightarrow{\tau} (\mathbf{skip}, ls[r := n])} \qquad \frac{x \in \text{Var}_G \quad a = \text{wr}^{[R]}(x, \llbracket E \rrbracket_{ls})}{(x :=^{[R]} E, ls) \xrightarrow{a} (\mathbf{skip}, ls)} \\
 \\
 \frac{a = \text{rd}^{[A]}(x, n) \quad n \in \text{Val}}{(r \leftarrow^{[A]} x, ls) \xrightarrow{a} (\mathbf{skip}, ls[r := n])} \qquad \frac{a = \text{upd}^{\text{RA}}(x, m, n) \quad m \in \text{Val}}{(x.\mathbf{swap}(n)^{\text{RA}}, ls) \xrightarrow{a} (\mathbf{skip}, ls)} \\
 \\
 \frac{(C_1, ls) \xrightarrow{a} (C'_1, ls')}{(C_1; C_2, ls) \xrightarrow{a} (C'_1; C_2, ls')} \qquad \frac{}{(\mathbf{skip}; C_2, ls) \xrightarrow{\tau} (C_2, ls)} \\
 \\
 \frac{\llbracket B \rrbracket_{ls}}{(\mathbf{if } B \mathbf{ then } C_1 \mathbf{ else } C_2, ls) \xrightarrow{\tau} (C_1, ls)} \qquad \frac{\neg \llbracket B \rrbracket_{ls}}{(\mathbf{if } B \mathbf{ then } C_1 \mathbf{ else } C_2, ls) \xrightarrow{\tau} (C_2, ls)} \\
 \\
 \frac{\llbracket B \rrbracket_{ls}}{(\mathbf{while } B \mathbf{ do } C, ls) \xrightarrow{\tau} (C; \mathbf{while } B \mathbf{ do } C, ls)} \qquad \frac{\neg \llbracket B \rrbracket_{ls}}{(\mathbf{while } B \mathbf{ do } C, ls) \xrightarrow{\tau} (\mathbf{skip}, ls)} \\
 \\
 \text{AUX} \frac{(A, ls) \xrightarrow{a} (\mathbf{skip}, ls') \quad (\alpha := E, ls') \xrightarrow{\tau} (\mathbf{skip}, ls'')}{(\langle A; \alpha := E \rangle, ls) \xrightarrow{a} (\mathbf{skip}, ls'')} \qquad \text{PROG} \frac{(P(t), lst(t)) \xrightarrow{a} (C, ls) \quad a \in \text{Act}_\tau}{(P, lst) \xrightarrow{a}_t (P[t := C], lst[t := ls])}
 \end{array}$$

■ **Figure 4** Program semantics.

value read and $\text{wrval}(a) \in \text{Val}$ be the value written. We let \mathbf{U} denote the update actions, and distinguish the sets $\mathbf{W}_R \supseteq \mathbf{U}$ (write release), $\mathbf{R}_A \supseteq \mathbf{U}$ (read acquire), \mathbf{W}_X (write relaxed) and \mathbf{R}_X (read relaxed). Finally, we define $\mathbf{R} = \mathbf{R}_A \cup \mathbf{R}_X$ (all reads) and $\mathbf{W} = \mathbf{W}_R \cup \mathbf{W}_X$ (all writes). Typically, we refer to the elements of \mathbf{W} as *writes*, but note that this set also includes update actions.

4.1 Program Semantics

In the program semantics, we assume a function $\text{lst} \in \text{Tid} \rightarrow (\text{Var}_L \leftrightarrow \text{Val})$ (\leftrightarrow being a partial function), which returns the local state for the given thread. We assume that the local variables of threads are disjoint, i.e., if $t \neq t'$, then $\text{dom}(\text{lst}(t)) \cap \text{dom}(\text{lst}(t')) = \emptyset$. For an expression E over local variables, we write $\llbracket E \rrbracket_{ls}$ for the value of E in local state $ls \in (\text{Var}_L \leftrightarrow \text{Val})$; we write $ls[r := n]$ to state that ls remains unchanged except for the value of local variable r which becomes n .

Figure 4 gives the transition rules of the program semantics. The last rule, **Prog**, lifts the transitions of threads to a transition for a concurrent program. The other rules concern the sequential part of the language. The rules in a sense ignore the fact that the language allows for global variables; the program semantics just details the values of local variables in component ls . When global variables are read, the program semantics allows for *any* possible value to be read. This is combined with the memory semantics (formalised by \xrightarrow{a}_t) as follows:

$$\frac{(P, lst) \xrightarrow{\tau}_t (P', lst')}{(P, lst, \sigma) \Longrightarrow (P', lst', \sigma)} \qquad \frac{(P, lst) \xrightarrow{a}_t (P', lst') \quad \sigma \xrightarrow{a}_t \sigma'}{(P, lst, \sigma) \Longrightarrow (P', lst', \sigma')}$$

The transitions defined by $\sigma \xrightarrow{a}_t \sigma'$ ensure that read actions only return a value allowed by the C11 semantics and are defined in Section 4.2. The rules for all imperative program constructs (sequential composition, **if** and **while**) are standard.

■ **Table 1** Components of a C11 state.

Component	Informal meaning	Initial value
$writes \subseteq W \times \mathbb{Q}$	The writes which have happened so far	$writes_{\mathbf{Init}}$
$tview_t \in Var_G \rightarrow writes$	The viewfront of a thread t	$tview_{\mathbf{Init}}$
$mview_w \in Var_G \rightarrow writes$	The viewfront of a thread when writing w	$mview_{\mathbf{Init}}$
$covered \subseteq writes$	The covered writes	\emptyset

4.2 Memory Semantics

Next, we detail the memory semantics, which is equivalent to an earlier operational reformulation [12] of the RAR fragment from [22].

C11 State. Table 1 summarises the components of a C11 state. Each global write is represented by a pair $(a, q) \in W \times \mathbb{Q}$, where a is a write action, and q is a rational number that we use as a *timestamp* (c.f., [16, 13, 27]). The timestamps totally order the writes to each variable; the ordering induced by timestamps is also referred to as the *modification order* [22, 12] or *coherence order* [6]. For each write $w = (a, q)$, we denote w 's timestamp by $tst(w) = q$. We also lift the functions var and $wval$ to timestamped writes, e.g., $var((a, q)) = var(a)$. The set of all writes that have occurred in the execution thus far is recorded in the state component $writes \subseteq W \times \mathbb{Q}$.

As described in Section 2, each state must record the writes that are observable to each read. To achieve this, we use two families of functions from global variables to writes, both of which record the *viewfronts* (c.f., [27, 17]).

- A function $tview_t$ that returns the *viewfront* of thread t (one for each global variable). The thread t can read from any write to variable x whose timestamp is not earlier than $tview_t(x)$. Accordingly, we define, for each state σ , thread t and global variable x , the set of *observable writes*:

$$\sigma.OW(t, x) = \{(a, q) \in \sigma.writes \mid var(a) = x \wedge tst(\sigma.tview_t(x)) \leq q\} \quad (1)$$

- A function $mview_w$ that records the *viewfront* of write w , which is set to be the viewfront of the thread that executed w at the time of w 's execution. We use $mview_w$ to compute a new value for $tview_t$ if a thread t *synchronizes* with w , i.e., if $w \in W_R$ and another thread executes an $e \in R_A$ that reads from w .

Finally, our semantics maintains a variable $covered \subseteq writes$. In C11 RAR, each update action occurs in modification order immediately after the write that it reads from [12]. This property constitutes the atomicity of updates. In order to preserve this property, we must prevent any newer write from intervening between any update and the write that it reads from. As we explain below, *covered* writes are those that are immediately prior to an update in modification order, and new write actions never interact with a covered write.

Initialisation. Table 1 also states how these components are initialised by **Init**. If $Var_G = \{x_1, \dots, x_n\}$, $Var_L = \{r_1, \dots, r_m\}$ and $k_1, \dots, k_n, l_1, \dots, l_m \in Val$, we assume $\mathbf{Init} = x_1 := k_1; \dots; x_n := k_n; [r_1 := l_1]; \dots [r_m := l_m];$, where we use the notation $[r_i := l_i]$ to mean that the assignment $r_i := l_i$ may optionally appear in **Init**. Thus each shared variable is initialised exactly once and each local variable is initialised at most once. The initial values of the state components are then as follows, where we assume that 0 is the initial timestamp.

$$\begin{aligned}
\text{writes}_{\mathbf{Init}} &= \{(wr(x_1, k_1), 0), \dots, (wr(x_n, k_n), 0)\} \\
\text{tview}_{\mathbf{Init}}(x_i) &= (wr(x_i, k_i), 0) \quad \text{for each thread } x_i \in \text{Var}_G \\
\text{mview}_{\mathbf{Init}} &= \text{tview}_{\mathbf{Init}}
\end{aligned}$$

The initial local state component of each thread must also be compatible with **Init**, i.e., for each t if $r_i \in \mathbf{dom}(lst(t))$ we have that $(lst(t))(r_i) = l_i$ provided $r_i := l_i$ appears in **Init**.

We let $lst_{\mathbf{Init}}$ be the local state compatible with **Init**, let $\sigma_{\mathbf{Init}}$ denote the initial state defined by **Init**, and define $\Gamma_{\mathbf{Init}} = (lst_{\mathbf{Init}}, \sigma_{\mathbf{Init}})$.

Transition semantics. The transition relation of our semantics for global reads and writes is given in Figure 5. Each transition $\sigma \xrightarrow{a}_t \sigma'$ is labelled by an action a and thread t . The premise of each rule must identify the write w that the action interacts with. This is made more precise below.

Read transition by thread t . Here we assume that

- a is either a relaxed or acquiring read to variable x ,
- (w, q) is a write to x that t can observe (i.e., $(w, q) \in \sigma.OW(t, x)$), and
- the value read by a is the value written by w .

Each read causes the viewfront of t to be updated. This is computed as follows. If the read synchronises with the write, then the thread's new view will be a combination of its existing view, and the view of that write. In particular, for each variable x the new view of x will be the later of either $\text{tview}_t(x)$ or $\text{mview}_w(x)$, in timestamp order. To express this, we use an operation that combines two views v_1 and v_2 , by constructing a new view that takes the later of the writes at each variable:

$$(v_1 \otimes v_2)(x) = \begin{cases} v_1(x) & \text{if } \text{tst}(v_2(x)) \leq \text{tst}(v_1(x)) \\ v_2(x) & \text{otherwise} \end{cases}$$

If w and a do not synchronise, then tview_t is simply updated to include the new write.

For illustration, consider the picture in Figure 6. The x-axis depicts the timestamps of the writes, the y-axis the variables x, y and z , which we assume are initialised by writes x_0, y_0 and z_0 , respectively. The orange line shows the view of a thread, say t_1 , and the blue line depicts the view of another thread that executes $w = (wr^R(y, 42), 3)$. If thread t_1 performs an acquiring read of y and reads from w (i.e., it performs a synchronising read), thread t_1 's view changes to the diagram on the right, whereby its current viewfront is combined with the viewfront of w .

Write transition by thread t . A write transition must identify the write (w, q) after which a occurs. This w must be observable and must *not* be covered – the second condition is required to preserve the read-modify-write atomicity of updates. We must choose a fresh timestamp $q' \in \mathbb{Q}$ for a , which is formalised by $\text{fresh}(q, q')$:

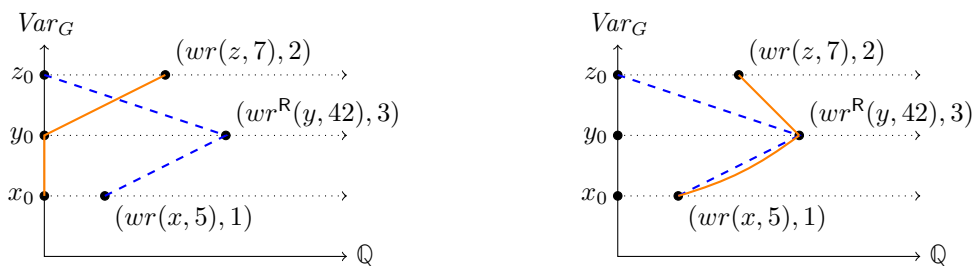
$$\sigma.\text{fresh}(q, q') = q < q' \wedge \forall w' \in \sigma.\text{writes}. q < \text{tst}(w') \Rightarrow q' < \text{tst}(w')$$

The predicate $\text{fresh}(q, q')$ ensures that q' is a new timestamp for the variable x , such that (a, q') occurs immediately after (w, q) ⁴. The new write is added to the set writes . We update tview_t to include the new write, which means t can no longer observe any writes prior to (a, q') .

⁴ This does not exclude that later some other write is placed in between q and q' .

$$\begin{array}{c}
a \in \{rd(x, n), rd^A(x, n)\} \quad (w, q) \in \sigma.OW(t, x) \quad wrval(w) = n \\
tview'_t = \begin{cases} \sigma.tview_t \otimes \sigma.mview_{(w, q)} & \text{if } (w, a) \in W_R \times R_A \\ \sigma.tview_t[x := (w, q)] & \text{otherwise} \end{cases} \\
\text{READ} \text{-----} \\
\sigma \xrightarrow{t} \sigma[tview_t := tview'_t] \\
\\
a \in \{wr(x, n), wr^R(x, n)\} \quad (w, q) \in \sigma.OW(t, x) \setminus \sigma.covered \quad \sigma.fresh(q, q') \\
writes' = \sigma.writes \cup \{(a, q')\} \quad tview'_t = \sigma.tview_t[x := (a, q')] \\
\text{WRITE} \text{-----} \\
\sigma \xrightarrow{t} \sigma[tview_t := tview'_t, mview_{(a, q')} := tview'_t, writes := writes'] \\
\\
a = upd^{RA}(x, m, n) \quad (w, q) \in \sigma.OW(t, x) \setminus \sigma.covered \\
wrval(w) = m \quad \sigma.fresh(q, q') \\
writes' = \sigma.writes \cup \{(a, q')\} \quad covered' = \sigma.covered \cup \{(w, q)\} \\
tview'_t = \begin{cases} \sigma.tview_t[x := (a, q')] \otimes \sigma.mview_{(w, q)} & \text{if } w \in W_R \\ \sigma.tview_t[x := (a, q')] & \text{otherwise} \end{cases} \\
\text{UPDATE} \text{-----} \\
\sigma \xrightarrow{t} \sigma[tview_t := tview'_t, mview_{(a, q')} := tview'_t, \\
writes := writes', covered := covered']
\end{array}$$

■ **Figure 5** Transition relation of the memory semantics.



■ **Figure 6** Illustration of views and view updates: pre-state (left) and post-state (right) after executing $rd^A(y, 42)$ by thread t_1 (orange).

Finally, we set the viewfront of (a, q') to be the new viewfront of t , i.e., $mview_{(a, q')} := tview'_t$. Now, if some other thread synchronises with this new write in some later transition, that thread's view will become at least as recent as t 's view at this transition.

Update transition by thread t . These transitions are best understood as a combination of the read and write transitions. As with a write transition, we must choose a valid fresh q' , and the state components $writes$ and $mview$ are updated in the same way. As discussed earlier, in UPDATE transitions it is necessary to record that the write that the update interacts with is now covered, which is achieved by adding that write to $covered$. Finally, we must compute a new thread view, which is similar to a READ transition, except that the thread's new view always includes the new write introduced by the update.

4.3 Relationship to the Axiomatic Semantics

We prove that the timestamp-based semantics presented here is equivalent to an earlier operational semantics [12] that is already known to be equivalent to the C11 RAR fragment. Here, we just roughly sketch how this proof proceeds.

The semantics in [12] describes C11 states in the form $E = (X, \text{sb}, \text{rf}, \text{mo})$, where X is a set of read and write events (roughly equivalent to actions) and sb , rf and mo describe the sequenced-before and reads-from relation as well as the modification order of the C11 axiomatic semantics. A number of further relations are derived from these, in particular the extended coherence order eco and the happens-before order hb . The proof of equivalence of the semantics shows the two semantics to *simulate* each other. For this, we need to define a correspondence between C11 states of form E and of form σ such that: (1) For $\sigma.\text{writes}$, we take $X \cap W$; (2) For $\sigma.\text{covered}$, we take the writes w in $X \cap W$ such that there is an update u with $(w, u) \in \text{rf}$; and (3) For mview and tview , we use a downward closure operator, \mathbf{cclose} , which for a given set of events S determines the set of events prior to S in the relation $\text{eco}^? \circ \text{hb}^?$ (where $R^?$ is the reflexive closure of a relation R). Then $\sigma.\text{tview}_t = \mathbf{max}_{\text{mo}}(X.\mathbf{cclose}(X_t))$ and $\sigma.\text{mview}_w = \mathbf{max}_{\text{mo}}(X.\mathbf{cclose}(\{w\}))$, where \mathbf{max}_{mo} selects writes being maximal wrt. mo and X_t are all actions of t in X . In all these cases, timestamps for writes have to be selected consistent with mo .

Given such a correspondence, the proof proceeds by showing this correspondence is preserved by the read, write and update transitions.

4.4 Well Formedness

Our proofs in subsequent sections require that the state under consideration is *well-formed*. This is formalised by predicate wfs over a C11 state σ , where

$$\begin{aligned} wfs(\sigma) \iff & \text{ran}((\bigcup_t \sigma.\text{tview}_t) \cup (\bigcup_w \sigma.\text{mview}_w)) \subseteq \sigma.\text{writes} \wedge \\ & \text{finite}(\sigma.\text{writes}) \wedge \sigma.\text{covered} \subseteq \sigma.\text{writes} \wedge \\ & (\forall w. w \in \sigma.\text{writes} \Rightarrow \sigma.\text{mview}_w(\text{var}(w)) = w) \end{aligned}$$

The first conjunct ensures that each viewable write is in $\sigma.\text{writes}$. The second conjunct ensures there are only a finite number of writes, and the third ensures that every covered write is an actual write. The final conjunct ensures that for each write in $\sigma.\text{writes}$, the viewfront of w for $\text{var}(w)$ is w itself.

Well-formedness is invariant for any program, i.e., every initialisation establishes well-formedness and every program transition preserves well-formedness.

► **Lemma 1.** *For any program C constructed using the syntax described in Section 3, $wfs(\sigma)$ is invariant.*

Proof. In Isabelle. We show that every initialisation establishes $wfs(\sigma)$. Furthermore, if $wfs(\sigma)$ and $\sigma \xrightarrow{a}_t \sigma'$, then $wfs(\sigma')$ for any action a and thread t . ◀

5 Hoare Logic and Owicki-Gries Reasoning for C11

In this section, we present a Hoare logic [15] for C11 RAR that enables Owicki-Gries reasoning [25]. For compound statements (including concurrent composition) we use the standard rules of Hoare logic as well as the standard interference freedom proof obligations described by Owicki and Gries. Our contribution is a novel set of high-level predicates that describe the *observations* of each thread for a C11 state, together with a set of *basic axioms* that describe how these predicates interact with read, write and update transitions. Soundness of these axioms has been checked using Isabelle.

In Section 5.1, we link our operational semantics to the proof outlines of Hoare logic and Owicki-Gries' notion of interference freedom. Section 5.2 provides an overview of our assertion language and briefly discusses the main categories of assertions, i.e., assertions

$$\begin{array}{c}
\text{SKIP} \frac{}{\{p\}\mathbf{skip}\{p\}} \quad \text{SEQ} \frac{\{p\}C_1\{r\} \quad \{r\}C_2\{q\}}{\{p\}C_1; C_2\{q\}} \\
\\
\text{IF} \frac{\{p \wedge B\}C_1\{q\} \quad \{p \wedge \neg B\}C_2\{q\}}{\{p\}\mathbf{if } B \mathbf{ then } C_1 \mathbf{ else } C_2\{q\}} \quad \text{WHILE} \frac{\{p \wedge B\}C\{p\}}{\{p\}\mathbf{while } B \mathbf{ do } C\{p \wedge \neg B\}} \\
\\
\text{UNTIL} \frac{\{p\}C\{r\} \quad \{r\}\mathbf{while } \neg B \mathbf{ do } C\{r \wedge B\}}{\{p\}\mathbf{do } C \mathbf{ until } B\{r \wedge B\}} \quad \text{CONS} \frac{p \Rightarrow p' \quad \{p'\}C\{q'\} \quad q' \Rightarrow q}{\{p\}C\{q\}}
\end{array}$$

■ **Figure 7** Classical proof rules for sequential programs.

describing observability, ordering and occurrences of writes. We present the basic axioms in stages, using specific litmus tests (in Sections 5.3, 5.4, 5.5) to motivate each group of assertions. The proof outlines of all litmus tests have been verified using Isabelle.

5.1 Soundness and Classical Verification Rules

We first define the meaning of a Hoare triple under partial correctness and present the classical proofs rules for compound statements. Unlike Hoare logic, where a state is modelled by a mapping from variables to values, as we have seen in Section 4.1, states of a C11 program contain two components: a local state lst and a global state σ . We let Σ_G be the set of all possible global state configurations (as described in Table 1) and let $\Sigma_{C11} = (Var_L \rightarrow Val) \times \Sigma_G$ be the set of all possible C11 states. Predicates over Σ_{C11} are therefore of type $\Sigma_{C11} \rightarrow \mathbb{B}$. This leads to the following definition of a Hoare triple, which we note is the same as the standard definition – the only difference is that the state component is of type Σ_{C11} .

► **Definition 2.** *Suppose $p, q \in \Sigma_{C11} \rightarrow \mathbb{B}$, $P \in Prog$ and $\mathbf{E} = \lambda t : Tid. \mathbf{skip}$. The semantics of a Hoare triple under partial correctness is given by:*

$$\begin{aligned}
\{p\}\mathbf{Init}\{q\} &= q(\Gamma_{\mathbf{Init}}) \\
\{p\}P\{q\} &= \forall lst, \sigma, lst', \sigma'. p(lst, \sigma) \wedge (P, lst, \sigma) \Longrightarrow^* (\mathbf{E}, lst', \sigma') \Rightarrow q(lst', \sigma') \\
\{p\}\mathbf{Init}; P\{q\} &= \exists r. \{p\}\mathbf{Init}\{r\} \wedge \{r\}P\{q\}
\end{aligned}$$

The classical rules of sequential Hoare logic for compound (i.e., non-atomic) statements are given in Figure 7. Soundness of these proof rules (with respect to Definition 2) holds for exactly the same reason as soundness of Hoare logic [15].

The sequential part is combined with the Owicki-Gries rule for concurrent composition in the standard way [25, 7]. First, we construct *proof outlines* for every component of the concurrent program in isolation. A proof outline inserts assertions (in $\{ \}$ brackets) into a program. In a so-called *standard* proof outline every statement R of the program has exactly one assertion before it. This assertion is its *precondition*, $pre(R)$. Next, all assertions in one component have to be checked for non-interference with all statements in other components.

► **Definition 3.** *A statement $R \in ACom$ with precondition $pre(R)$ (in the standard proof outline) does not interfere with an assertion p if*

$$\{p \wedge pre(R)\} R \{p\} .$$

11:12 Owicki-Gries Reasoning for C11 RAR

Proof outlines of concurrent programs are interference free if no statement in one thread interferes with an assertion in another thread.

Interference freedom guarantees that proof outlines in each thread are stable under the execution of other threads. This is formalised in the Owicki-Gries proof rule for concurrent composition:

$$\text{PARALLEL} \frac{\text{Proof outlines } \{p_i\}C_i\{q_i\} \text{ are interference free}}{\{\bigwedge_{i=1}^n p_i\} C_1 \parallel \dots \parallel C_n \{\bigwedge_{i=1}^n q_i\}}$$

We say a proof outline is *valid* if it is both sequentially valid (or locally correct) and interference free.

Finally, there is a standard proof rule for auxiliary variables in parallel programs [7]. Let V be a set of auxiliary variables of a parallel program P and q be a predicate that does not mention auxiliary variables. Then we can prove that a Hoare triple holds for a program extended with auxiliary variables and transfer this proof to the original program:

$$\text{AUXVAR} \frac{\{true\} \mathbf{Init}; P \{q\}}{\{true\} \mathbf{Init}_0; P_0 \{q\}} \text{provided } \text{vars}(q) \cap V = \emptyset$$

where \mathbf{Init}_0 is obtained from \mathbf{Init} by removing all auxiliary assignments and P_0 is obtained by replacing all statements $\langle A, a := E \rangle$ in P (for $a \in V$) by A .

5.2 An Assertion Language

We studied a number of well-known litmus tests and examples and discovered three main categories of assertions required for specification and verification of a wide range of problems. These three main categories are dealing with (values of) writes to variables and the order in which they occur.

- **Observability.** Observability assertions describe if or when a thread may observe or has encountered a write to a variable. As described in Section 2, these assertions are thread-specific and deal with the thread's view. We repeat the main ideas here to simplify comparison with the other types of assertions. The main observability assertions are as follows:
 1. **Possible observation** which is denoted by $x \approx_t u$ means that thread t *may* observe value u for x . The formal definition and an example motivating this assertion is given in Section 5.4.
 2. **Definite observation** which is denoted by $x =_t u$ means that thread t *must* observe the value u for x . The formal definition and an example motivating this assertion is given in Section 5.3.
 3. **Conditional observation** which is denoted by $[x = u](y =_t v)$ means that if thread t synchronises with a write to variable x with value u , it *must* observe value v for y . The formal definition and an example motivating this assertion is given in Section 5.4.
 4. **Encountered value** which is denoted by $x \stackrel{enc}{=} _t v$ means that thread t has encountered (had the opportunity to observe) a write to variable x with value v . The formal definition and three examples motivating this assertion are given in Section 5.5.
- **Ordering.** Ordering assertions specify the order of values written to a variable by different writes. These assertions are thread-independent and specify an order over the timestamp of various writes with specific values:

1. **Possible value order** which is denoted by $m \prec_x n$ means that there exists two writes w and w' to variable x where the timestamp of w' is larger than the timestamp of w and the value of w and w' is m and n , respectively.
2. **Definite value order** which is denoted by $m \ll_x n$ means that for all writes w and w' to x where the value of w is m and the value of w' is n , the timestamp of w' is larger than the timestamp of w and $m \prec_x n$.

Both the above assertions are formally defined in Section 5.5 and examples showing their usage are provided.

- **Occurrence.** Occurrence assertions specify the occurrence of a write with a specific value to a variable (regardless of observability). Similar to the previous category, these assertions are thread-independent:
 1. **Value occurrence** assertions specify the limit of occurrence of writes to a variable with a specific value. For instance, $\mathbb{0}_x n$ means that no write with value n to variable x has occurred or $\mathbb{1}_x n$ means that there is *at most* one write with value n to x in the current state. The formal definition and examples of these assertions are given in Section 5.5.
 2. **Initial value** which is denoted by $x_{\text{Init}} = n$ means that the initial value written to x is n . The formal definition and examples of this assertion are also given in Section 5.5.
 3. **Covered write** assertions, denoted by \mathbf{C}_x^n , state that all writes to variable x except the last write are covered by an update (see Section 4.2), and that the last write to x has value n . This assertion is formally defined in Section 6 and is used in verification of Peterson's mutual exclusion algorithm.

5.3 Load Buffering

Our first example is the load buffering litmus test (see Figure 8), which we can show satisfies the postcondition $r1 = 0 \vee r2 = 0$ since our semantics assumes absence of cycles in the sequence-before relation combined with reads-from [22, 12]. The assertions about the C11 state capture properties about *definite observations* (i.e., observability assertions), which we formalise below.

For a set of writes W and variable $x \in \text{Var}_G$, let $W_x = \{w \in W \mid \text{var}(w) = x\}$ be the set of writes in W that write to x . We define the *last write* to x in W as:

$$\text{last}(W, x) = w \iff w \in W_x \wedge (\forall w' \in W_x. \text{tst}(w') \leq \text{tst}(w))$$

Moreover, we define the definite observation of a view function, *view* with respect to a set of writes as follows:

$$\text{dview}(\text{view}, W, x) = n \iff \text{view}(x) = \text{last}(W, x) \wedge \text{wrval}(\text{last}(W, x)) = n$$

The first conjunct ensures that the viewfront of *view* for x is the last write to x in W , and the second conjunct ensures that the value written by the last write to x in W is n .

Definite observation. For a variable x , thread t and value n , we define:

$$x =_t n = \lambda \sigma. \text{dview}(\sigma.\text{tview}_t, \sigma.\text{writes}, x) = n$$

Expanding this out, we obtain:

$$\sigma.\text{tview}_t(x) = \text{last}(\sigma.\text{writes}, x) \wedge \text{wrval}(\text{last}(\sigma.\text{writes}, x)) = n$$

$$\begin{array}{c}
\mathbf{Init}: x := 0; y := 0; r1 := 0; r2 := 0; \\
\{x =_1 0 \wedge y =_2 0 \wedge r1 = 0 \wedge r2 = 0\} \\
\begin{array}{c|c}
\mathbf{Thread 1} & \mathbf{Thread 2} \\
\{y =_2 0 \wedge r2 = 0\} & \{x =_1 0 \wedge r1 = 0\} \\
1 : r1 \leftarrow x; & 3 : r2 \leftarrow y; \\
\{y =_2 0 \wedge r2 = 0\} & \{x =_1 0 \wedge r1 = 0\} \\
2 : y := 1; & 4 : x := 1; \\
\{r1 = 0 \vee r2 = 0\} & \{r1 = 0 \vee r2 = 0\} \\
\{r1 = 0 \vee r2 = 0\} & \{r1 = 0 \vee r2 = 0\}
\end{array}
\end{array}$$

■ **Figure 8** Proof outline for load buffering.

The first conjunct ensures that the viewfront of t for x is the last write to x in σ (thus t can only read this last write to x). The second conjunct ensures that the value written by the last write is n . The function $dview$ is also used in the definition of conditional observation in Section 5.4.

The proof of load buffering relies on the basic axioms in the following lemma. We assume $atoms(\mathbf{Init})$ returns the set of assignments contained within \mathbf{Init} .

► **Lemma 4.** *Each of the basic axioms below is sound (as per Definition 2), where the statements are decorated with the thread identifier of the executing thread.*

$$\begin{array}{c}
\text{INIT} \frac{x := n \in atoms(\mathbf{Init})}{\{true\} \mathbf{Init} \{x =_t n\}} \quad \text{DOPRES-RD} \frac{}{\{x =_{t'} m\} r \leftarrow_t^{[A]} y \{x =_{t'} m\}} \\
\text{DOPRES-WR} \frac{x \neq y}{\{x =_{t'} n\} y :=_t m \{x =_{t'} n\}}
\end{array}$$

Proof. In Isabelle. ◀

Thus by rule INIT an assignment $x := n$ in INIT ensures that $x =_t n$ for all threads t holds at program start. Note that such an initial assertion for the entire program is not subject to non-interference checks. The rule DOPRES-RD states that a definite observation $x =_{t'} m$ is invariant over a read step executed by thread t . Note that pre/post conditions for DOPRES-RD refer to thread t' , while the read statement refers to thread t . Also note that there is no additional restriction on t and t' , thus the rule applies regardless of whether $t = t'$, or not. Similarly, there are two global variables x and y mentioned in the rule, but there are no further restrictions on their values. Rule DOPRES-WR gives a condition for invariance of a definite observation assertion over a write. It requires that the variable being observed is different from the variable that is updated.

► **Theorem 5.** *The proof outline for load buffering in Figure 8 is valid.*

Proof. The proof has been established in Isabelle. We outline the main steps below as it is instructive to understand the high-level proof strategy. First we establish local correctness:

- The initial condition is established by rule INIT, which is in turn used to establish the initial assertions in both threads.
- In thread 1, local correctness of the postcondition of line 1 (precondition of line 2) follows from rule DOPRES-RD, and the postcondition of line 2 follows by weakening. The proof of local correctness in thread 2 is symmetric.

We now establish interference freedom. The precondition of line 1 is interference free wrt line 3 by DOPRES-RD, and wrt line 4 by DOPRES-WR. This argument also applies to the precondition of line 2. Interference freedom of the postcondition of line 2 is trivial. The proof of interference freedom of the assertions in thread 2 is symmetric. ◀

5.4 Message Passing

Next we return to the message passing example from Section 2. Its verification requires the usage of the other two observability assertions.

Possible observation. For a variable x , thread t and value n , we define:

$$x \approx_t n = \lambda\sigma. \exists w \in \sigma.OW(t, x). \text{wval}(w) = n$$

Thus, there is a write to x that is observable to thread t with a value n .

Conditional observation. For variables x, y , thread t and values m, n , we define:

$$[x = n](y =_t m) = \lambda\sigma. \forall w \in \sigma.OW(t, x). \text{wval}(w) = n \Rightarrow \\ \text{act}(w) \in \mathbb{W}_R \wedge \text{dview}(\sigma.\text{mview}_w, \sigma.\text{writes}, y) = m$$

The antecedent assumes that the value read for x is n , and the consequent ensures that w is a releasing write such that the definite view of this write for variable y returns m . As we shall see, one useful way of establishing this condition is by falsifying the antecedent by ensuring that thread t cannot observe n for x (see (4) below).

Some useful relationships between the assertions above are given by the lemma below.

► **Lemma 6.** For variables $x, y \in \text{Var}_G$, thread t and values $m, n \in \text{Val}$, each of the following holds:

$$\text{wfs} \wedge x =_t n \Rightarrow x \approx_t n \quad (2)$$

$$\text{wfs} \wedge x =_t n \wedge x \approx_t m \Rightarrow n = m \quad (3)$$

$$x \not\approx_t n \Rightarrow [x = n](y =_t m) \quad (4)$$

$$x =_t n \wedge x =_{t'} m \Rightarrow n = m \quad (5)$$

Proof. In Isabelle. ◀

By (2), given a well-formed state any definite observation implies a possible observation, and by (3) a definite observation must agree with a possible observation. By (4) if it is not possible to observe the antecedent of a conditional observation, then the conditional observation must hold. By (5) any two definite value observations must agree (since they both observe the last write to x).

The next lemma lists the basic axioms that are used to prove correctness of the message passing example.

► **Lemma 7.** Each of the following rules is sound (as per Definition 2), where the statements are decorated with the thread identifier of the executing thread.

$$\text{MODLAST} \frac{}{\{x =_t n\} x :=_t m \{x =_t m\}} \quad \text{MODSOME} \frac{}{\{\text{true}\} x :=_t m \{x \approx_t m\}}$$

$$\text{NPOPRES} \frac{}{\{x \not\approx_t m\} r \leftarrow_{t'}^{[A]} y \{x \not\approx_t m\}} \quad \text{NOOW} \frac{x \neq y}{\{x \not\approx_t n\} y :=_{t'} m \{x \not\approx_t n\}}$$

$$\text{READLAST} \frac{}{\{x =_t m\} r \leftarrow_t x \{r = m\}}$$

$$\text{CO-INTRO} \frac{x \neq y}{\{y =_t m \wedge x \not\approx_{t'} n\} x :=_t^R n \{[x = n](y =_{t'} m)\}}$$

$$\text{TRANSFER} \frac{}{\{[x = n](y =_t m)\} r \leftarrow_t^A x \{r = n \Rightarrow y =_t m\}}$$

Proof. In Isabelle. ◀

► **Theorem 8.** *The proof outline of message passing in Figure 3 is valid.*

Proof. The proof has been established in Isabelle. We outline the main steps below. First we show local correctness.

- Using INIT we establish the precondition $f =_1 0 \wedge f =_2 0 \wedge d =_1 0 \wedge d =_2 0$.
- The precondition of the program implies the initial assertions of both threads. In thread 1, we use (3) to establish $f \not\approx_2 1$ since (3) is logically equivalent to

$$wfs \wedge x =_t n \wedge n \neq m \Rightarrow x \not\approx_t m$$

In thread 2, we use (3) in combination with (4).

- In thread 1, the post condition of line 1 (precondition of line 2) follows by application of NOOW and MODLAST. The post condition of line 2 is trivial.
- In thread 2, the postcondition of line 3 follows by application of TRANSFER, while the postcondition of line 4 follows by application of READLAST.

Next we show interference freedom.

- The preconditions of lines 1 and 2 can be shown to be interference free by applying NPOPRES to the first conjunct and DOPRES-RD to the second.
- The precondition of line 3 is interference free against line 1 due to NOOW using the existing precondition $f \not\approx_2 1$ of line 1. The proof then follows by application of (4). Interference freedom against line 2, is proved using CO-INTRO and the precondition at line 2.
- The precondition of line 4 is interference free against line 1 by (5) (i.e., since the preconditions of lines 1 and 4 are contradictory). Interference freedom holds against line 2 by rule DOPRES-WR.
- The postconditions of lines 2 and 4 are trivially interference free. ◀

5.5 Read-Read Coherence

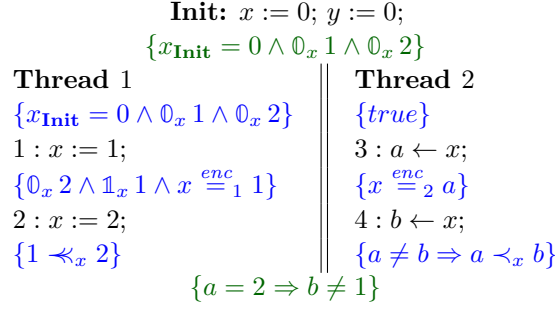
Next, we verify three versions of the read-read coherence (RRC) litmus test as given in Figures 9, 10 and 11. The original RRC litmus test (Figure 10) guarantees that if one thread sees the writes to x (by threads 1 and 2) in a certain order, then the other thread see the writes in the same order. Here, the postcondition assumes that thread 3 has observed the write $x := 1$, then the write $x := 2$, while thread 4 has already seen the write $x := 2$ when reading x at line 5. It requires that thread 4 does not subsequently see value 1 when it reads x at line 6. Figure 9 presents a simpler variation where the ordering of writes to x is enforced by the thread ordering. Figure 11 combines RRC with message passing.

Unlike message passing (which is a litmus test over two different variables), the RRC examples demonstrate the need for *ordering* and *occurrence* assertions which we introduce next.

Possible value order. For values m, n and variable x , we define:

$$m \prec_x n = \lambda\sigma. \exists w, w' \in \sigma.writes_x. wrval(w) = m \wedge wrval(w') = n \wedge tst(w) < tst(w')$$

Thus, there are two writes to x with values m and n , where the timestamp of the write with value m precedes the timestamp of the write with value n . Note that this $m \prec_x n$ does not preclude $n \prec_x m$. E.g., if a thread writes m to x , then n , then m again, both $m \prec_x n$ and $n \prec_x m$ will hold. In this scenario, $m \prec_x m$ also holds since there are two separate writes to x with value m .



■ **Figure 9** Proof outline for RRC2, where $x \in Var_G$ and $a, b \in Var_L$.

Definite value order. For values m, n and variable x , we define:

$$\begin{aligned}
m \prec_x n &= \lambda\sigma. (m \prec_x n)(\sigma) \wedge (\forall w, w' \in \sigma.writes_x. \\
&\quad wrval(w) = m \wedge wrval(w') = n \Rightarrow \\
&\quad tst(w) < tst(w'))
\end{aligned}$$

Note that this implies $m \neq n$. Unlike possible value orders if $m \prec_x n$ holds then $n \not\prec_x m$. Note also that our definition allows several writes to x with values m and n provided all writes with value m occur (in timestamp order) before all writes with value n .

Initial value. For values n and variable x , we define:

$$\begin{aligned}
x_{\text{Init}} = n &= \lambda\sigma. \exists w \in \sigma.writes_x. wrval(w) = n \wedge \\
&\quad (\forall w' \in \sigma.writes_x. w \neq w' \Rightarrow tst(w) < tst(w'))
\end{aligned}$$

Note that for the construction in this paper, it suffices to return the write to x with timestamp 0 since we assume that writes are initialised with timestamp 0. The definition above however, is more robust since it also applies to situations where variables are not initialised, or initialised to an arbitrarily chosen timestamp (as is the case in our Isabelle encoding).

Encountered value. For a variable x , thread t and value n , we define:

$$x \stackrel{enc}{=}_t n = \lambda\sigma. \exists w \in \sigma.writes_x. tst(w) \leq tst(\sigma.tview_t(x)) \wedge wrval(w) = n$$

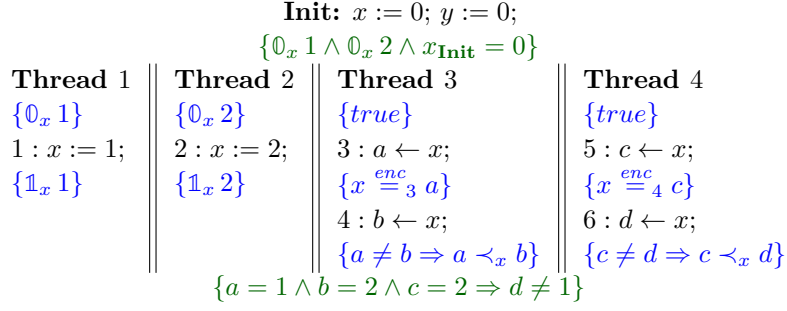
That is $x \stackrel{enc}{=}_t n$ holds iff there is a write to x with value n whose timestamp is at most the timestamp of the viewfront of t for x . Note that $x \stackrel{enc}{=}_t n$ does not guarantee that t has read the value n for x . For instance, $x \stackrel{enc}{=}_t n$ could hold if there is a write, say w , of x with value n and t writes to x with a write whose timestamp is greater than $tst(w)$.

Value occurrence. These are straightforward to define in terms of our value order assertions above. For a variable x , thread t and value n , we define:

$$\begin{aligned}
0_x n &= \exists m. x_{\text{Init}} = m \wedge m \neq n \wedge m \not\prec_x n \\
1_x n &= n \not\prec_x n
\end{aligned}$$

Thus, if $0_x n$ holds then there is no write with value n . If $1_x n$ holds, then either there is no write to x with value n , or if there is a write with value n , this is the only such write.

To understand the interaction between value ordering and write limit assertions, consider the following lemma. It states that if there is a possible value order on x with m preceding n and there is at most one write with these values, then there is a definite value order on x with m preceding n .



■ **Figure 10** Proof outline for RRC, where $x \in Var_G$ and $a, b, c, d \in Var_L$.

► **Lemma 9.** For $x \in Var_G$ and $m, n \in Val$, we have:

$$m \prec_x n \wedge 1_x m \wedge 1_x n \Rightarrow m \ll_x n \quad (6)$$

$$m \ll_x n \Rightarrow n \not\prec_x m \quad (7)$$

Proof. In Isabelle. ◀

We discuss the proof of RRC2 in detail. Its proof relies on the following lemma which captures some basic properties about value assertions.

► **Lemma 10.** Each of the rules below is sound (as per Definition 2), where the statements are decorated with the thread identifier of the executing thread.

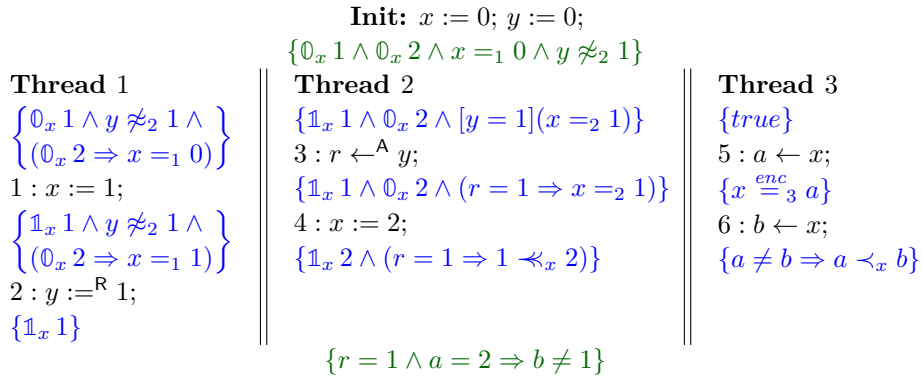
$$\begin{array}{c}
 \text{ZWR} \frac{m \neq n}{\{0_x m\} y := \overset{[R]}{t} n \{0_x m\}} \quad \text{DVPRES} \frac{}{\{m \ll_x n\} r \leftarrow \overset{[A]}{t} y \{m \ll_x n\}} \\
 \\
 \text{1INTRO} \frac{i \neq m}{\{x_{\text{Init}} = i \wedge 0_x m\} x := \overset{[R]}{t} m \{1_x m\}} \quad \text{ENCWR} \frac{}{\{true\} x := \overset{[R]}{t} m \{x \stackrel{enc}{=} m\}} \\
 \\
 \text{ENCRD} \frac{}{\{true\} r \leftarrow \overset{[A]}{t} x \{x \stackrel{enc}{=} r\}} \quad \text{EPO} \frac{}{\{x \stackrel{enc}{=} m\} r \leftarrow \overset{[A]}{t} x \{r \neq m \Rightarrow m \prec_x r\}} \\
 \\
 \text{DVINTRO} \frac{i \neq n}{\{x_{\text{Init}} = i \wedge 0_x n \wedge 1_x m \wedge x \stackrel{enc}{=} m\} x := \overset{[R]}{t} n \{m \ll_x n\}} \\
 \\
 \text{1PRESR} \frac{}{\{1_x m\} r \leftarrow \overset{[A]}{t} y \{1_x m\}} \quad \text{PORR} \frac{}{\{m \prec_x n\} C \{m \prec_x n\}}
 \end{array}$$

Proof. In Isabelle. ◀

► **Theorem 11.** The proof outline for RRC2 in Figure 9 is valid.

Proof. This proof has been mechanised in Isabelle. Once again, we describe the proof outline to give an overview of how our proofs are used. For local correctness we have the following.

- The initialisation clearly satisfies the precondition of the program, and this implies the precondition of thread 1. The precondition of thread 2 is trivial.
- Next we consider the postcondition of line 1. The first conjunct holds by ZWR, the second conjunct holds by 1INTRO and the third by rule ENCWR.
- The postcondition of line 2 holds by rule DVINTRO.



■ **Figure 11** Proof outline for RRC3, where $x, y \in Var_G$ and $a, b \in Var_L$.

- In thread 2, the postcondition of line 3 holds by rule ENCRD, and the postcondition of line 4 holds by rule EPO.

Next we check interference freedom.

- The precondition of line 1 is stable with respect to lines 3 and 4 by ZWR.
- Next consider the precondition of line 2. The first and second conjuncts are stable with respect to lines 3 and 4 by ZWR and 1PRESR, respectively. The third conjunct is trivially preserved (see Isabelle).
- The postcondition of line 2 holds by DVPRES.
- The precondition of line 3 is trivial and the postcondition of line 3 holds by PORD. ◀

Correctness of RRC and RRC3 is established by the following theorem.

► **Theorem 12.** *The proof outlines for RRC and RRC3 in Figure 10 and Figure 11, respectively are valid.*

Proof. In Isabelle. ◀

For RRC (Figure 10), the precondition of line 4 records the fact that thread 3 has encountered a (whatever the value of a may be). Moreover, it guarantees that there is at most one write of x with values 1 and 2. The first conjunct (i.e., $x \stackrel{enc}{=} a$) allows us to conclude that after x is read at line 4, if a and b are different, then the value for a is possibly ordered before the value for b . The second and third conditions are used to establish the postconditions $1_x 1$ and $1_x 2$. This argument also applies to the assertions in thread 4. Finally, we show that the postcondition of the program holds as follows, where we assume $post$ is the conjunction of the postcondition of each thread.

$$\begin{aligned}
 & post \Rightarrow (a = 1 \wedge b = 2 \wedge c = 2 \Rightarrow d \neq 1) \\
 \iff & post \wedge a = 1 \wedge b = 2 \wedge c = 2 \wedge d = 1 \Rightarrow false && \text{(logic)} \\
 \iff & 1_x 1 \wedge 1_x 2 \wedge 1 \prec_x 2 \wedge 2 \prec_x 1 \Rightarrow false && \text{(logic)} \\
 \iff & 1 \prec_x 2 \wedge 2 \prec_x 1 \Rightarrow false && (6) \\
 \iff & true && (7)
 \end{aligned}$$

The calculation above has been verified with Isabelle, but we recall the proof here as it provides insight into the interactions between different value assertions.

RRC3 (Figure 11) combines message passing on y with RRC on x . Namely, knowledge of $x := 1$ in thread 1 is transferred to thread 2 using a release-acquire synchronisation on y . Thus, if thread 2 reads 1 for y it must also have encountered 1 for x . Thus, if $r = 1$,

then the write on line 4 must have happened *after* the write on line 1. This means that it should be impossible for thread 3 to read 2 for x (at line 5) then read 1 for x (at line 6). Unlike message passing, in RRC3, the “data” variable x is updated both before and after synchronisation. Thus, the assertions on definite values (e.g., $x =_1 1$) become conditional on whether line 4 has already been executed. In particular, the antecedent $\mathbb{0}_x 2$ allows us to assume that line 4 has not yet been executed. As with RRC, we must separately prove that the conjunction of the postconditions of the threads implies the postcondition of the program. This proof is mechanised in Isabelle, and is elided here.

6 Case study: Peterson’s algorithm

We turn to our final case study, the verification of the mutual exclusion property of a version of Peterson’s algorithm. The complexity of this case study is much greater than our earlier examples. This program contains a loop, features a careful mixture of relaxed and release/acquire operations to the same variable, and an RMW operation whose precise semantics is critical to the correctness of the algorithm.

Our version of Peterson’s algorithm⁵, presented in Figure 12 is a mutual exclusion algorithm for two threads implemented for C11 using release-acquire annotations [34]. The purpose of verification is to show that this algorithm actually guarantees mutual exclusion, i.e., that the two threads can never be in their critical sections (line 6) at the same time. As with the original algorithm, variable $flag_i$, for $i \in \{1, 2\}$ is used to indicate whether thread i intends to enter its critical section. In this version of the algorithm, we let $flag_i$ range over $\{0, 1\}$, where 0 is used for the boolean value “false”, and 1 is used for the boolean value “true”. The shared variable $turn$ is used to cause a thread to “give way” when both threads intend to enter their critical sections at the same time. Our verification uses auxiliary variables $after_i$ for each thread i (as does the proof for a sequentially consistent setting in [7]), the purpose of which we describe below.

We describe the algorithm for thread 1; the other thread is symmetric. For now, we ignore the assertions. The flag variable is set to 1 (line 1) using a relaxed write (which cannot induce any synchronisation), but is set to 0 (line 7) using a release annotation. The intention of the latter is to synchronise this write (of 0 to $flag_1$) with the read of $flag_1$ at line 3 in thread 2. The value of $turn$ is set using a **swap** command. The **swap** is implemented using an C11 RMW operation that has both the release and acquire annotations. When the **swap** is executed, as part of the same transition, the auxiliary variable $after_1$ is also set, indicating that thread 1 is ready to enter the busy wait loop beginning at line 3, and then to enter the critical section.

The busy wait loop forces thread 0 to wait until either $flag_2$ is 0 (indicating that thread 2 is not trying to enter the critical section) or $turn = 1$ (indicating that it is thread 1’s turn to enter the critical section). Note that the read of $turn$ within the guard of the busy wait loop (line 5) is relaxed.

We turn now to the proof that this version of Peterson’s algorithm has the mutual exclusion property. We prove mutual exclusion in two steps. First, we show that the given proof outline is valid, and second, that the conjunction of the precondition of thread 1’s critical section (line 6) and thread 2’s must be false. Therefore, the two threads cannot simultaneously be in their critical sections.

⁵ For simplicity our version of the algorithm does not have an outermost loop.

Init: $flag_1 := 0; flag_2 := 0; turn := 0 \wedge after_1 := false; after_2 := false$

Thread 1

$$\left\{ \begin{array}{l} \neg after_1 \wedge flag_1 =_1 0 \wedge turn \not\approx_2 2 \wedge (C_{turn}^0 \vee [turn = 1](flag_2 =_1 1)) \\ \wedge (after_2 \Rightarrow C_{turn}^1 \wedge [turn = 1](flag_2 =_1 1)) \end{array} \right\}$$

1: $flag_1 := 1$;

$$\left\{ \neg after_1 \wedge flag_1 =_1 1 \wedge turn \not\approx_2 2 \wedge (after_2 \Rightarrow C_{turn}^1 \wedge [turn = 1](flag_2 =_1 1)) \right\}$$

2: $(turn.swap(2)^{RA} ; after_1 := true)$

$$\left\{ after_1 \wedge (after_2 \wedge (flag_2 \approx_1 0 \vee turn \approx_1 1) \Rightarrow turn =_2 1) \right\}$$

do

3: $r_1 \leftarrow^A flag_2$

$$\left\{ after_1 \wedge (after_2 \wedge (r_1 = 0 \vee turn \approx_1 1 \vee flag_2 \approx_1 0) \Rightarrow turn =_2 1) \right\}$$

4: $r_2 \leftarrow turn$

$$\left\{ after_1 \wedge (after_2 \wedge (r_1 = 0 \vee r_2 = 1 \vee turn \approx_1 1 \vee flag_2 \approx_1 0) \Rightarrow turn =_2 1) \right\}$$

5: **until** $(r_1 = 0 \vee r_2 = 1)$

$$\left\{ after_1 \wedge (after_2 \Rightarrow turn =_2 1) \right\}$$

6: Critical section ;

7: $(flag_1 :=^R 0 ; after_1 := false)$

■ **Figure 12** Peterson’s algorithm (adapted from [34]) and its proof outline. **Thread 2** (not shown) is symmetric.

We deal with the second step first by showing that the formula below is *false*:

$$after_1 \wedge (after_2 \Rightarrow turn =_2 1) \wedge after_2 \wedge (after_1 \Rightarrow turn =_1 2)$$

It is easy to see that this implies $turn =_1 2 \wedge turn =_2 1$. However, by (5) this situation is impossible.

The first step is more elaborate and we only describe certain aspects. The precondition of line 3 is also an invariant of the busy wait loop. This assertion ensures that if thread 1 is able to exit the busy wait loop, then the precondition of the critical section will be satisfied. Note that thread 1 exits the loop if it reads 0 from $flag_2$ (which is only possible when $flag_2 \approx_1 0$) or it reads 1 from $turn$ (which is only possible when $turn \approx_1 1$). The invariant states that if one of these conditions holds in a state where thread 2 is waiting to enter the critical section (that is, $after_2$), we can conclude $turn =_2 1$ as required.

Proving that the precondition of line 3 is satisfied in the post-state of line 2 requires using a feature of our assertion language, closely related to the semantics of RMW operations, that we now introduce. Recall from the UPDATE rule in Figure 5 that whenever a write w is read-from by an RMW operation, w becomes *covered*, so that no later write (or RMW) operation can be inserted between w and the RMW. This feature of C11 is critical to the correctness of Peterson’s algorithm. Observe that the $turn$ variable is only modified by RMW operations, and therefore every write to $turn$ is covered, except the last. To formally state this, we need the third *occurrence* assertion C_x^n , defined as follows.

$$C_x^n = \lambda\sigma. \forall w \in \sigma.writes_x. w \notin \sigma.covered \Rightarrow wrval(w) = n \wedge w = last(W, x)$$

So C_x^n means that every write to x except the last is covered and the value written by that last write is n .

We use the following lemma on covered.

► **Lemma 13.**

$$\begin{array}{c}
 \text{CVD-UPD} \frac{}{\{\mathbf{C}_x^n\} \ x.\text{swap}(l)^{RA} \ \{\mathbf{C}_x^l\}} \qquad \text{CVD-WR} \frac{x \neq y}{\{\mathbf{C}_x^n\} \ y := [R] \ m \ \{\mathbf{C}_x^n\}} \\
 \text{CVD-RD} \frac{}{\{\mathbf{C}_x^n\} \ r \leftarrow [A] \ y \ \{\mathbf{C}_x^n\}} \qquad \text{CVD-DOBS} \frac{}{\{\mathbf{C}_x^n\} \ x.\text{swap}(l)^{RA} \ \{x =_t \ l\}}
 \end{array}$$

Rule CVD-UPD states that if \mathbf{C}_x^n holds in the pre-state, then after executing $x.\text{swap}(l)^{RA}$, we obtain a new covered predicate \mathbf{C}_x^l . Thus, it is possible to maintain a covered predicate in a program (with possibly different return values) by ensuring each modification to the covered variable is via a swap. This is a property that is true of Peterson’s algorithm as given in Figure 12. Rules CVD-WR and CVD-RD give preservation properties for the covered assertion for a read and a write, respectively. Finally, CVD-DOBS is used to establish a definite observation of a covered assertion after a swap command.

The precondition of line 2 asserts that if thread 2 is ready to enter the critical section (that is, after_2) then the RMW to be executed at line 2 must read from the last write which has value 1 (that is, \mathbf{C}_{turn}^1) and when this RMW occurs then thread 1 will definitely see flag_2 set (that is, $[turn = 1](\text{flag}_2 =_1 1)$). This is enough to show that if after_2 then in the post-state of the RMW, $\text{flag}_2 \not\approx_1 0$ which is sufficient to prove the postcondition of line 2.

Of course, the sequential reasoning above must be combined with an interference freedom check, which is supported by a set of basic lemmas describing how \mathbf{C}_x^n is updated. This leads to the following theorem, which establishes validity of the proof outline.

► **Theorem 14.** *The proof outline of Peterson’s algorithm (Figure 12) is valid.*

Proof. In Isabelle. ◀

We note that Peterson’s algorithm represents a challenge in deductive verification. Unlike the litmus tests presented above, there is sufficient complexity in the algorithm and the resulting proof outline so that pen-and-paper proofs cannot be trusted. Using our mechanisation, we explored several variations of the proof outline in Figure 12, and discovered simplifications to our original pen-and-paper proofs.

7 Mechanisation

As already mentioned, the operational semantics as well as all lemmas and theorems presented in this paper have been mechanised in Isabelle. In this section, we discuss our mechanisation effort.

To prove the lemmas about basic assertions, we typically prove a more general result relating to reads and writes, which are then specialised so that they can be used in the verification of the algorithms. For example, we first prove the lemma in Figure 13, which describes changes to definite values and applies to any writing transition. This is then specialised to the corollaries on the right, which are easier for Isabelle to find when performing the verification of the proof outlines.

The generic lemmas require some amount of interactive work. However, once verified, it is straightforward to use them to prove the corollaries. For example, `d_obs_WrX_set` in Figure 13 is verified using “`by (metis WrX_def avar.simps(2) d_obs_Wr_set wr_val.simps(1))`”, which is found automatically by Isabelle’s built in `sledgehammer` tool [10].

Such lemmas and corollaries are in turn used in the proofs of programs. First the program state (i.e., Σ_{C11}) is encoded as a `record` type with a special variable that models the C11 state. The programs themselves are encoded as a relation over these records with program

```

lemma d_obs_Wr_set:
  assumes "wfs  $\sigma$ "
    and "wr_val a = Some n"
    and "avar a = x"
    and "[x =t m]  $\sigma$ "
    and "step t a  $\sigma$   $\sigma'$ "
  shows "[x =t n]  $\sigma'$ "

corollary d_obs_WrX_set:
  "wfs  $\sigma \implies [x =_t m] \sigma \implies \sigma [x := n]_t \sigma' \implies [x =_t n] \sigma'"

corollary d_obs_WrR_set :
  "wfs  $\sigma \implies [x =_t m] \sigma \implies \sigma [x :=^R n]_t \sigma' \implies [x =_t n] \sigma'"

corollary d_obs_RMW_set :
  "wfs  $\sigma \implies [x =_t m] \sigma \implies \sigma \text{RMW}[x,w,n]_t \sigma' \implies [x =_t n] \sigma'"$$$ 
```

■ **Figure 13** Isabelle encoding of basic axioms over C11 assertions.

counters modelling control flow. This allows the proof outlines to be encoded as predicates mapping program counters to the assertions at that control point. We then verify a set of lemmas that guarantee local correctness and interference freedom, where we decompose proofs and apply case analysis over the individual program steps (e.g., reads, writes for each thread). Once a proof has been decomposed, `sledgehammer` is able to find the relevant corollaries (e.g., those in Figure 13) to discharge proofs automatically.

8 Related Work

The semantics and verification of programs running on weak memory models has recently received a lot of attention. Lahav [20] gives a brief survey for C11.

Our timestamp based operational semantics is motivated by ideas in [13] and is similar to the semantics of Kaiser et al. [16, 17]. We note there are differences in coverage of the memory models in [13, 16, 17]. Dolan et al. [13] cover a sequentially consistent (SC) and relaxed accesses for OCAML, where the SC operations behave like Java volatiles. Kaiser et al [16] covers non-atomics and release-acquire, while Kang et al. [17] support a much larger fragment of C11, including so-called load-buffering cycles.

Abdulla et al. have shown the reachability problem for release-acquire to be *undecidable* [1]. A number of works target *model checking* for weak memory, e.g., by explicitly encoding architectural structures leading to weak behaviour, like store buffers [31, 4]. Ponce de León et al. [28, 14] have developed a bounded model checker for weak memory models, taking the axiomatic description of a memory model as input. (Bounded) model checkers for specific weak memory models are furthermore the tools CBMC [5] (for TSO), NIDHUGG [2] (for TSO and PSO), RCMC [18] (for C11) and GENMC [19] (again, parametric in memory model).

A (non-automatic) reasoning technique for proving invariants – parameterised by a weak memory model – has been proposed by Alglave and Cousot [3]. They propose a new semantics, different from an operational one without any coherence order (or modification order) constraining the order of writes to memory. Their assertions contain so-called pythia variables to uniquely identify values of read events, and require a separate communication

proof (differentiating their method from standard Owicki-Gries reasoning). They say “In addition to the initialisation, sequential, and non-interference proof, the main difference with Owicki and Gries [25] (and Lamport 1977) is the use of pythia variables and the read-from relation in assertions and the communication proof showing that reads-from is well-formed.” [3]. Our method in contrast only requires the initialisation, sequential, and non-interference proofs as with the original technique.

Another manual method for the RC11 memory model has been developed by Doherty et al. [12], who cover the message passing example and Peterson’s algorithm. Our work is inspired by this existing work, however, there are several differences. They use a classical model of the C11 state (expressed in terms of a set of relations, e.g., reads-from, sequenced-before etc), develop assertions over these relations and a small proof calculus for these assertions. Moreover, their methods are at a lower level of abstraction than the techniques presented in this paper since the assertions are stated in terms of individual relations that make up each state. Thus, it is not possible to directly develop a Hoare logic for their assertions and mechanisation itself is more difficult.

Also close to our work is that of Lahav and Vafeiadis [21] who also develop an Owicki-Gries style proof calculus. We consider all their examples except RCU – our logic can handle the RCU example, but this proof has thus far not been mechanised. Moreover, we include several other case studies such as litmus tests that combine read-read coherence with message passing and the non-trivial Peterson’s algorithm. There are several additional differences to note. (1) Lahav and Vafeiadis’ proof calculus is developed in the absence of an operational semantics, and hence, their definition of a valid Hoare triple is non standard (see [21, Definition 9]). A consequence of this is that they must be careful about the introduction of auxiliary variables, resorting to the more restricted notion of a *ghost* variable. In contrast, we use traditional auxiliary variables – an auxiliary variable must not affect the control flow of a program nor be assigned to any program variable. Note however, that to simplify the presentation, we use auxiliary variables in a more restricted manner (see Section 3). (2) They do not handle relaxed accesses – as stated in their conclusion: “While OGRA’s non-interference condition appears to be restrictive, it is unsound for weaker memory models, such as C11’s relaxed accesses ...”. (3) They do not provide a mechanisation.

A frequently employed starting point for program logic is separation logic, for which a number of extensions to weak memory exist (GPS [32], RSL [16]). Svendsen et al. [30] propose a separation logic based on the promising semantics of Kang et al. [17]. The principle of ownership transfer used therein naturally fits to message passing using release acquire. Prover support for such separation logic based proofs – like ours with Isabelle – has been developed for the Iris proof system [16]. Tool support has also been developed by Summers and Müller [29], where the RSL logic has been encoded in the Viper tool, offering a level of proof automation. Their encoding is proved sound and complete with respect to RSL. However, such efforts do not provide a clear link between C11 semantics and traditional reasoning using Hoare logics.

9 Conclusion

In this paper, we have introduced an assertion language for C11 RAR which enables re-use of the entire Owicki-Gries proof calculus except for the axiom of assignment. The assertion language is based on an operational semantics for C11 RAR which we have shown to be sound wrt. standard axiomatic semantics. We have exemplified reasoning on a number of standard C11 RAR litmus tests as well as a C11 RAR annotated version of Peterson’s algorithm. All

proofs ranging are mechanised within Isabelle – this includes soundness of the basic axioms for weak memory reads, writes and updates, and validity of proof outlines for the examples presented.

We are currently integrating this work [11] into the standard Owicki-Gries library that is included in the Isabelle distribution [24]. As future work, we aim to tackle fragments of C11 larger than C11 RAR, e.g., fragments that allow the load buffering example to terminate with postcondition $r1 = 1 \wedge r2 = 1$ [8, 17], SC annotations [22], as well as release sequences and fences [9]. Extending our operational semantics to handle the final two features is straightforward, but is not considered in this paper as it complicates the semantics and detracts from our main contribution, i.e., a simple extension to Hoare logic to enable reasoning about C11 programs. Hoare-style reasoning that incorporates the other two features is currently being investigated.

References

- 1 P. A. Abdulla, J. Arora, M. F. Atig, and S. N. Krishna. Verification of programs under the release-acquire semantics. In K. S. McKinley and K. Fisher, editors, *PLDI*, pages 1117–1132. ACM, 2019.
- 2 P. Aziz Abdulla, S. Aronis, M. Faouzi Atig, B. Jonsson, C. Leonardsson, and K. Sagonas. Stateless model checking for TSO and PSO. *Acta Inf.*, 54(8):789–818, 2017. doi:10.1007/s00236-016-0275-0.
- 3 J. Alglave and P. Cousot. OGRE and Pythia: an invariance proof method for weak consistency models. In G. Castagna and A. D. Gordon, editors, *POPL*, pages 3–18. ACM, 2017.
- 4 J. Alglave, D. Kroening, V. Nimal, and M. Tautschnig. Software verification for weak memory via program transformation. In M. Felleisen and P. Gardner, editors, *ESOP*, volume 7792 of *LNCS*, pages 512–532. Springer, 2013. doi:10.1007/978-3-642-37036-6_28.
- 5 J. Alglave, D. Kroening, and M. Tautschnig. Partial orders for efficient bounded model checking of concurrent software. In N. Sharygina and H. Veith, editors, *CAV*, volume 8044 of *LNCS*, pages 141–157. Springer, 2013. doi:10.1007/978-3-642-39799-8_9.
- 6 J. Alglave, L. Maranget, and M. Tautschnig. Herding cats: Modelling, simulation, testing, and data mining for weak memory. *ACM Trans. Program. Lang. Syst.*, 36(2):7:1–7:74, 2014.
- 7 K. R. Apt, F. S. de Boer, and E.-R. Olderog. *Verification of Sequential and Concurrent Programs*. Texts in Computer Science. Springer, 2009. doi:10.1007/978-1-84882-745-5.
- 8 M. Batty, A. F. Donaldson, and J. Wickerson. Overhauling SC atomics in C11 and OpenCL. In *POPL*, pages 634–648. ACM, 2016.
- 9 M. Batty, S. Owens, S. Sarkar, P. Sewell, and T. Weber. Mathematizing C++ concurrency. In T. Ball and M. Sagiv, editors, *POPL*, pages 55–66. ACM, 2011. doi:10.1145/1926385.1926394.
- 10 S. Böhme and T. Nipkow. Sledgehammer: Judgement day. In *IJCAR*, volume 6173 of *Lecture Notes in Computer Science*, pages 107–121. Springer, 2010.
- 11 S. Dalvandi, B. Dongol, and S. Doherty. Integrating Owicki-Gries for C11-style memory models into Isabelle/HOL. *CoRR*, abs/2004.02983, 2020. arXiv:2004.02983.
- 12 S. Doherty, B. Dongol, H. Wehrheim, and J. Derrick. Verifying C11 programs operationally. In J. K. Hollingsworth and I. Keidar, editors, *PPoPP*, pages 355–365. ACM, 2019. doi:10.1145/3293883.3295702.
- 13 S. Dolan, K. C. Sivaramakrishnan, and A. Madhavapeddy. Bounding data races in space and time. In *PLDI*, PLDI 2018, pages 242–255, New York, NY, USA, 2018. ACM.
- 14 N. Gavrilenko, H. Ponce de Le'on, F. Furbach, K. Heljanko, and R. Meyer. BMC for weak memory models: Relation analysis for compact SMT encodings. In I. Dillig and S. Tasiran, editors, *CAV*, volume 11561 of *LNCS*, pages 355–365. Springer, 2019. doi:10.1007/978-3-030-25540-4_19.

- 15 C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, 1969. doi:10.1145/363235.363259.
- 16 J.-O. Kaiser, H.-H. Dang, D. Dreyer, O. Lahav, and V. Vafeiadis. Strong logic for weak memory: Reasoning about release-acquire consistency in Iris. In P. Müller, editor, *ECOOP*, volume 74 of *LIPICs*, pages 17:1–17:29. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2017. doi:10.4230/LIPICs.ECOOP.2017.17.
- 17 J. Kang, C.-K. Hur, O. Lahav, V. Vafeiadis, and D. Dreyer. A promising semantics for relaxed-memory concurrency. In G. Castagna and A. D. Gordon, editors, *POPL*, pages 175–189. ACM, 2017.
- 18 M. Kokologiannakis, O. Lahav, K. Sagonas, and V. Vafeiadis. Effective stateless model checking for C/C++ concurrency. *PACMPL*, 2(POPL):17:1–17:32, 2018. doi:10.1145/3158105.
- 19 M. Kokologiannakis, A. Raad, and V. Vafeiadis. Model checking for weakly consistent libraries. In K. S. McKinley and K. Fisher, editors, *PLDI*, pages 96–110. ACM, 2019.
- 20 O. Lahav. Verification under causally consistent shared memory. *SIGLOG News*, 6(2):43–56, 2019. doi:10.1145/3326938.3326942.
- 21 O. Lahav and V. Vafeiadis. Owicki-Gries reasoning for weak memory models. In M. M. Halldórsson, K. Iwama, N. Kobayashi, and B. Speckmann, editors, *ICALP*, volume 9135 of *LNCS*, pages 311–323. Springer, 2015. doi:10.1007/978-3-662-47666-6_25.
- 22 O. Lahav, V. Vafeiadis, J. Kang, C.-K. Hur, and D. Dreyer. Repairing sequential consistency in C/C++11. In *PLDI*, pages 618–632. ACM, 2017.
- 23 L. 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.
- 24 T. Nipkow and L. P. Nieto. Owicki/Gries in Isabelle/HOL. In *FASE*, volume 1577 of *Lecture Notes in Computer Science*, pages 188–203. Springer, 1999.
- 25 S. S. Owicki and D. Gries. An axiomatic proof technique for parallel programs I. *Acta Inf.*, 6:319–340, 1976. doi:10.1007/BF00268134.
- 26 L. C. Paulson. *Isabelle - A Generic Theorem Prover (with a contribution by T. Nipkow)*, volume 828 of *LNCS*. Springer, 1994. doi:10.1007/BFb0030541.
- 27 A. Podkopaev, I. Sergey, and A. Nanevski. Operational aspects of C/C++ concurrency. *CoRR*, abs/1606.01400, 2016. arXiv:1606.01400.
- 28 H. Ponce de León, F. Furbach, K. Heljanko, and R. Meyer. BMC with memory models as modules. In N. Bjørner and A. Gurfinkel, editors, *FMCAD*, pages 1–9. IEEE, 2018. doi:10.23919/FMCAD.2018.8603021.
- 29 A. J. Summers and P. Müller. Automating deductive verification for weak-memory programs. In D. Beyer and M. Huisman, editors, *TACAS*, volume 10805 of *LNCS*, pages 190–209. Springer, 2018. doi:10.1007/978-3-319-89960-2_11.
- 30 K. Svendsen, J. Pichon-Pharabod, M. Doko, O. Lahav, and V. Vafeiadis. A separation logic for a promising semantics. In A. Ahmed, editor, *ESOP*, volume 10801 of *LNCS*, pages 357–384. Springer, 2018. doi:10.1007/978-3-319-89884-1_13.
- 31 O. Travkin, A. Mütze, and H. Wehrheim. SPIN as a linearizability checker under weak memory models. In V. Bertacco and A. Legay, editors, *HVC*, volume 8244 of *LNCS*, pages 311–326. Springer, 2013. doi:10.1007/978-3-319-03077-7_21.
- 32 A. Turon, V. Vafeiadis, and D. Dreyer. GPS: navigating weak memory with ghosts, protocols, and separation. In A. P. Black and T. D. Millstein, editors, *OOPSLA*, pages 691–707. ACM, 2014. doi:10.1145/2660193.2660243.
- 33 J. Wickerson, M. Batty, T. Sorensen, and G. A. Constantinides. Automatically comparing memory consistency models. In G. Castagna and A. D. Gordon, editors, *POPL*, pages 190–204. ACM, 2017.
- 34 A. Williams. https://www.justsoftwaresolutions.co.uk/threading/petersons_lock_with_C++0x_atomics.html, 2018. Accessed: 2018-06-20.

A Semantics for the Essence of React

Magnus Madsen 

Aarhus University, Denmark

<https://www.cs.au.dk/~magnus/>


magnusm@cs.au.dk

Ondřej Lhoták 

University of Waterloo, Canada

<https://plg.uwaterloo.ca/~olhotak/>

olhotak@uwaterloo.ca

Frank Tip 

Northeastern University, Boston, MA, USA

<https://www.franktip.org>

f.tip@northeastern.edu

Abstract

Traditionally, web applications have been written as HTML pages with embedded JavaScript code that implements dynamic and interactive features by manipulating the Document Object Model (DOM) through a low-level browser API. However, this unprincipled approach leads to code that is brittle, difficult to understand, non-modular, and does not facilitate incremental update of user-interfaces in response to state changes.

React is a popular framework for constructing web applications that aims to overcome these problems. React applications are written in a declarative and object-oriented style, and consist of components that are organized in a tree structure. Each component has a set of properties representing input parameters, a state consisting of values that may vary over time, and a render method that declaratively specifies the subcomponents of the component. React's concept of reconciliation determines the impact of state changes and updates the user-interface incrementally by selective mounting and unmounting of subcomponents. At designated points, the React framework invokes lifecycle hooks that enable programmers to perform actions outside the framework such as acquiring and releasing resources needed by a component.

These mechanisms exhibit considerable complexity, but, to our knowledge, no formal specification of React's semantics exists. This paper presents a small-step operational semantics that captures the essence of React, as a first step towards a long-term goal of developing automatic tools for program understanding, automatic testing, and bug finding for React web applications. To demonstrate that key operations such as mounting, unmounting, and reconciliation terminate, we define the notion of a well-behaved component and prove that well-behavedness is preserved by these operations.

2012 ACM Subject Classification Theory of computation → Operational semantics; Software and its engineering → Semantics

Keywords and phrases JavaScript, React, operational semantics, lifecycle, reconciliation

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2020.12

Related Version <https://cs.uwaterloo.ca/sites/ca.computer-science/files/uploads/files/cs-2020-03.pdf>.

Funding The third author was supported in part by National Science Foundation grant CCF-1715153. This research was supported by the Natural Sciences and Engineering Research Council of Canada.

Acknowledgements The authors are grateful to the anonymous reviewers for their insightful comments.



© Magnus Madsen, Ondřej Lhoták, and Frank Tip;

licensed under Creative Commons License CC-BY

34th European Conference on Object-Oriented Programming (ECOOP 2020).

Editors: Robert Hirschfeld and Tobias Pape; Article No. 12; pp. 12:1–12:26

Leibniz International Proceedings in Informatics



LIPICs Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

1 Introduction

A web application is a program where the user-interface runs in a web browser. Traditionally, such applications have been written as HTML pages that contain embedded JavaScript code that implements dynamic and interactive features, such as input validation or data visualization, by manipulating the Document Object Model (DOM) through a low-level browser API. Using the DOM, the programmer can add, remove, or mutate HTML elements directly. While expressive, this unprincipled approach has several disadvantages. First, direct mutation of the DOM leads to brittle and difficult to understand code. Second, using this approach, it is difficult to design reusable user-interface components and libraries. Third, this approach does not easily lend itself to designs where the user-interface of a web application is updated *incrementally* in response to user input or new data received from a server. As a result, traditional web applications are often buggy and difficult to maintain [4, 5, 19, 20, 18].

The React framework [8] was developed to address these concerns. A React application does not manipulate the DOM directly but instead operates on a “virtual DOM”, by constructing React components that are rendered incrementally as their properties and state change. Such components are written in a declarative and object-oriented programming style, where classes represent components, and reusing a component is as simple as creating an instance of a class. A *React application* is structured as a tree, where a root component represents the top-level element of the user-interface, and where a (possibly dynamically varying) set of subcomponents correspond to widgets within that page. A *React component* has three key constituents: (i) a set of *properties* representing input parameters needed to configure the component, (ii) an internal *state* consisting of values that may vary over time, and (iii) a **render** method that specifies how a component is rendered by returning a subtree composed of a mix of subcomponents and HTML elements. The process of creating and updating the user-interface of a React application is defined in terms of *mounting* and *unmounting* operations, corresponding to the addition and removal of subcomponents. A key feature of React is its concept of *reconciliation*, which entails determining those parts of a page that are affected by state changes and updating them incrementally by selectively mounting and unmounting subcomponents. At key points during this process (e.g., when components are mounted or unmounted), the React framework invokes *lifecycle* hooks – callback methods that enable programmers to perform actions, e.g., to fetch data from a remote server or to store data locally in `localStorage`.

Today, React is one of the world’s most popular web frameworks. On StackOverflow, a popular question-and-answer forum for programmers, more than 181,554 questions are tagged `reactjs`. In comparison, the `reactjs` tag is more popular than the `perl`, `scala`, `swing`, or `typescript` tags. On GitHub, React is the fourth most starred repository, with more than 142,000 stars. On NPM, the package manager for Node.js, React has more than 20,000,000 downloads per month.

While React helps programmers structure their web application as a collection of modular components, it comes with its own set of challenges and bug patterns that new programmers must learn to avoid. For example, the intricate control- and dataflow can make it exceedingly difficult to understand how state changes in one component affect other components. As another example, the complex interplay between the component lifecycle methods and the reconciliation algorithm can be difficult to understand.

To enable the construction of tools for reasoning about the behavior of React applications, for automatic testing, and for bug finding, a precise understanding of the semantics of React is required. This paper establishes such an understanding, in the form of a formal semantics that captures the essence of React. Our semantics is based on λ_{js} [10], and precisely models the key aspects of React:

- (i) mounting and unmounting of components,
- (ii) reconciliation of component descriptors and mounted components, and
- (iii) the semantics of state changes.

To demonstrate that key operations such as mounting, unmounting, and reconciliation terminate, we define the notion of a *well-behaved* component by imposing a ranking function on components, and requiring that the `render` method of a component only returns components of strictly smaller rank. We then prove that well-behavedness is preserved by these operations.

In this paper, we focus on the core of React version 16.x. In the 16.x series, React has undergone some changes in the supported lifecycle methods, but those are mostly orthogonal to our work. React 16.8 introduced *React hooks*, a new, optional mechanism for state management in a functional style that avoids the use of classes. To our knowledge, there is no plan to change or remove the current mechanism for state management, and it is unclear to what extent the community will be adopting React hooks. In the paper, we focus on traditional state management as used in current React applications.

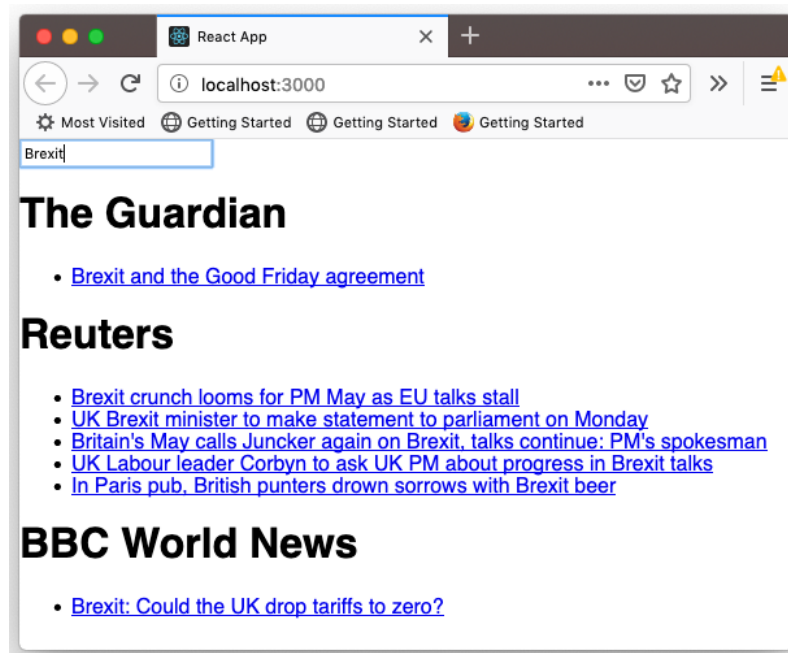
The remainder of this paper is organized as follows. Section 2 uses a small React application as an example to illustrate the key concepts and terminology associated with React. Section 3 presents a small-step operational semantics for the essence of React. Section 4 defines a well-behavedness property for components and demonstrates that well-behavedness is preserved by the key operations. Section 5 discusses how lifecycle hooks can be modeled. Related work is discussed in Section 6. Finally, Section 7 presents conclusions and directions for future work.

2 React

We will review the key concepts and terminology of React using a small React application that illustrates some of the typical requirements that a modern web developer must deal with. This includes fetching data and periodically receiving updates from the server, updating the browser’s Document Object Model (DOM) to reflect the latest data, and filtering data based on user input. In pure JavaScript these steps can be difficult to manage, but React makes these steps easy to express.

Our example application receives RSS feeds from several news sites and, for each feed, displays the title of each news article. Clicking on the title will navigate the user to the full article on newspaper website. To focus on a specific news topic, the user may enter a keyword in the box at the top of the window to remove from the view any news items that do not contain the specified keyword. Figure 1 shows a screenshot of the application after the user has typed the word “brexit” in the box. The news feeds are polled every 5 seconds and the display is updated when existing news items disappear, and when additional news items appear. Note that such updates are performed *incrementally*, i.e., only the changed parts of the web page’s DOM representation are updated and re-rendered.

In general, a *React application* is organized as a tree of *React components*, each of which is self-contained UI widget that may be composed of *subcomponents*. React components are either instances of classes or they are HTML elements such as buttons or text fields. Our example application consists of a *root component* `App` that has 4 subcomponents: a text field and one subcomponent for each news feed, which is an instance of the `RssFeed` class. Each React component has three central constituents: A set of *properties*, an internal *state*, and a `render` method. The properties are a form of input parameters typically used to configure the component. The state holds time-varying values, e.g., the values of input fields. The `render` method is used to draw the component by returning a subtree composed of a mix of



■ **Figure 1** Screenshot of our example React application. The screenshot shows the set of articles from news feeds from The Guardian, Reuters, and BBC World News after the user has entered the search term “Brexit”.

subcomponents and HTML elements. In the case of our example application, the number of subcomponents is fixed because it depends on the number of news feeds being monitored, which is fixed. However, in general the component tree is not static, as a `render` method can vary the tree returned based on a component’s properties and state. For example, one can easily imagine adding a feature that would allow the user to subscribe to additional news feeds, so that the number of components would vary dynamically as well.

The process of creating and updating the user-interface of a React application is defined in terms of *mounting* and *unmounting* operations. Here, mounting an application involves instantiating the class corresponding to its root component, rendering it by calling its `render` method, and recursively mounting its subcomponents. The React framework automatically takes care of all of this. To do so, React must be informed explicitly when state changes occur, by invoking the `setState` method with an object that specifies the state changes. When state changes occur, React will invoke the `render` method of the affected components to update the user-interface appropriately. However, changes are applied *incrementally*: React’s *reconciliation* mechanism ensures that state changes do not require recomputation and re-rendering of the entire component tree, but only of those components affected by the state change. If a state change has the effect of removing a subcomponent, such a subcomponent is *unmounted*, i.e., the subcomponent is removed from its parent, and cleanup actions are performed as necessary. At designated points in the execution of a React application (e.g., prior to and upon completion of mounting and unmounting operations and when state changes occur), so-called *lifecycle methods* are invoked by the React framework. Lifecycle methods are declared in classes corresponding to React components and can perform any programmer-specified action. Typically, lifecycle methods are used to initiate network requests to fetch data or initialize resources when a component is mounted, and to free resources when a component is unmounted.

```

1  import React, {Component} from 'react';
2  import './App.css';
3
4  let Parser = require('rss-parser');
5  let parser = new Parser();
6
7  class App extends Component {
8    constructor() {
9      super();
10     this.state = {filter: ""};
11   }
12   feeds = [
13     { title: "The Guardian", url: "https://www.theguardian.com/world/zimbabwe/rss" },
14     { title: "Reuters", url: "http://feeds.reuters.com/Reuters/worldNews"},
15     { title: "BBC World News", url: "http://feeds.bbc.co.uk/news/world/rss.xml"}
16   ];
17   render = () => {
18     return <div>
19       <input type="text" onChange={this.notifyChange}/>
20       { this.feeds.
21         map((feed) => <RssFeed title={feed.title} url={feed.url} filter={this.state.filter}/>)}
22     </div>
23   }
24   notifyChange = (e) => {
25     this.setState({filter: e.target.value});
26   }
27 }
28 class RssFeed extends Component {
29   constructor() {
30     super();
31     this.state = {items: []};
32   }
33   componentWillMount = () => {
34     this.doUpdate()
35     this.timer = setInterval(this.doUpdate, 5000)
36   }
37   componentWillUnmount = () => {
38     clearInterval(this.timer)
39   }
40   doUpdate = () => { // use "cors-anywhere" proxy to add CORS headers to the proxied request
41     (async () => {
42       let url = "https://cors-anywhere.herokuapp.com/" + this.props.url
43       let feed = await parser.parseURL(url);
44       this.setState({items: feed.items});
45     })();
46   }
47   matchesKeyword = (newsItem) =>
48     (this.props.filter === "") || newsItem.title.includes(this.props.filter);
49   render = () => {
50     return (
51       <div className="feed">
52         <h1>{this.props.title}</h1>
53         <ul>
54           {this.state.items.filter(this.matchesKeyword).
55             map(item => <li><a href={item.link}>{item.title}</a></li>)}
56         </ul>
57       </div>
58     );
59   }
60 }
61 export default App;

```

■ **Figure 2** Example of a React web application.

Figure 2 shows the complete source code for the React application shown in Figure 1. The application consists of two classes: `App` (lines 7–27) and `RSSFeed` (lines 28–60). Each React class has a constructor method that is responsible for initializing the component’s state. The state of the root component `App` includes a field¹ `filter` that will be used to determine which items should be selected from the newsfeeds. The constructor of the root component `App` (line 8–11) initializes the `filter` field to the empty string, reflecting that, by default, no news

¹ In this paper, we will use the term “field” to refer to object fields and the term “property” to refer to the inputs provided to React components.

items should be filtered out. Next, on lines 12–16, a field `feeds` is initialized with an array that contains the URLs for the news feeds. Lines 17–23 show the `render` method for class `App`. In general, a `render` method of a React class produces a *component descriptor*, i.e., a declarative description of the component’s subcomponents that are to be mounted/reconciled by React. Here, the `render` method returns a `<div>` tag containing a text field (line 19), and a sequence of `RSSFeed` components (lines 20–21). Line 19 specifies that entering or changing the text in the text field will cause the method `notifyChange` (lines 24–26) to be invoked with the current text as an argument. Lines 20–21 map a function over the `feeds` array to create an array of `RSSFeed` components, passing each field’s URL and title, and the current value of the `filter` property. The `notifyChange` method on lines 24–26 specifies that React’s `setState` function should be invoked, passing in an object with a property `filter` that is bound to the current value of the filter. This illustrates how React merges a form of declarative and object-oriented programming: The `render` methods return a declarative description of subcomponents that React then instantiates and maintains as objects.

The state of an `RSSFeed` component consists of a field `items` that represents the current items of the corresponding news feed, and the constructor (lines 29–32) initializes this field with an empty array. Next, the lifecycle methods `componentWillMount` (lines 33–36) and `componentWillUnmount` (lines 37–39) are defined. The former specifies that, when an `RSSFeed` component mounts, the `doUpdate` method should be invoked (line 34) immediately, and then invoked periodically (line 35) every 5 seconds. The latter specifies that, upon unmounting an `RSSFeed` component, the timer should be cleared (line 38). The `doUpdate` method (lines 40–46) asynchronously requests content from an RSS feed on line 42, using a proxy to enable cross-origin requests that would otherwise be disallowed due to browser’s same-origin policy. The contents of the feed are parsed on line 43, and the `RSSFeed` component’s state is updated on line 44 by invoking `setState` to set the `items` property to the news items in the feed’s contents. The `RSSFeed` component’s `render` method (lines 49–59) makes use of an auxiliary method `matchesKeyword` (lines 47–48) to determine if a given newsfeed item matches the filter if a filter is specified (if no filter is specified, all items match). The `render` method returns a `div` element (line 51) containing a title (line 52) and a list of news items extracted from the feed (line 55). The latter is constructed by filtering the `items` using the `matchesKeyword` function and creating an `li` (list item) tag for each item containing a hyperlink that is created using the title and URL obtained from the news feed.

We conclude from this example that React enables the construction of sophisticated interactive web applications, for which the user-interface is modular and incrementally maintained in response to property and state changes. React applications are remarkably concise due to a powerful combination of declarative and object-oriented programming. While the behavior of React applications can be understood in terms of a small number of key operations, thus far these operations have only been defined informally. To our knowledge, our paper presents the first approach that places the essence of React on a formal foundation.

3 Semantics

We now present a small-step operational semantics, named λ_{react} , that captures the essence of React. We formulate the semantics as an extension of the λ_{js} calculus [10]. Using λ_{js} as a foundation allows us to focus on the core issues without being distracted by complex JavaScript features such as prototype-based inheritance, dynamic property access, implicit coercions, and so on, which are handled by λ_{js} .

The calculus aims to capture three central aspects of React:

- (i) the mounting and unmounting of components,
- (ii) the reconciliation of component descriptors and mounted components, and
- (iii) the semantics of state changes.

3.1 Design Decisions

We briefly outline the major design choices we made in the formulation of the semantics.

- React is a huge framework and our aim to distill it down to its essence. We want to describe the primary concepts of React in as little formalism as possible. It is *not* our goal to provide a complete formal specification of the entire React framework.
- React relies on classes, which are supported in EcmaScript 6, but λ_{js} is based on an earlier version of EcmaScript. We believe that λ_{js} could be extended with classes, but that is beyond the scope of the current paper. In the λ_{react} semantics, we side-step this issue by direct modeling of the React constructs.
- We extend the syntax of λ_{js} with explicit terms for mounting, unmounting, and reconciliation of components. In React, the programmer cannot use these terms directly; they are part of the internals.
- We model the registration of event listeners since they are the main driver of execution once a React application has started. We simulate the execution of these event listeners in a non-deterministic fashion with a special “•” term that represents the event loop.
- React places a strong emphasis on performance. For the most part, we ignore such considerations, however our specification of object equivalence and merging does reflect these underlying concerns.
- We omit lifecycle hooks from the λ_{react} semantics. Although they play an important role in any realistic React application they are not particularly interesting from a semantic point of view, and adding them would be straightforward, if tedious. Nevertheless, in Section 5 we give some ideas of how to incorporate lifecycle hooks into the semantics.

3.2 Components, Component Descriptors, and Mounted Components

A React component is a class that extends the `React.Component` class. Each React component has a set of *properties* and an internal *state*. The properties are a form of input parameters used to configure the component, whereas the state holds time-varying values.

Every component has a `render` method that returns fragments that are either React *component descriptors* or HTML elements. This tree fragment represents the “view” of the component and is used by React to “draw” the component in the DOM. For example, a component could return the HTML element `<h1>Hello</h1>`, which React would simply display. On the other hand, it could also return a component descriptor `<RssFeed title="..." url="..." />` whose view would depend on the `render` method inside `RssFeed`.

Component Descriptors and Mounted Components

In React terminology, the process of creating a React component is called *mounting*. A mounted component is an object that is currently part of the virtual (and real DOM). When a component is taken out of the virtual DOM (and real DOM) it is *unmounted* and eventually garbage-collected. A *component descriptor* is tag-like structure that carries a name of a class and optionally several properties. A component descriptor can be turned into a mounted component. To illustrate these concepts, consider the following `render` method:

12:8 A Semantics for the Essence of React

```
62 class App extends React.Component {
63   render() {
64     if (this.state.progress < 100) {
65       return <ProgressBar value={this.state.progress} />
66     } else {
67       return <Game />
68     }
69   }
70   ...
71 }
```

Here the `render` method consists of an if-then-else statement. If the current progress, kept in the internal state of the `App` component, is less than 100 then the method returns the component descriptor `<ProgressBar value=... />` passing the current progress as a property. Otherwise, the method returns the `<Game />` component descriptor.

When the `App` component is mounted, its initial progress is zero. Hence the render method returns the `<ProgressBar />` descriptor. React then mounts and displays this component. Over time, the progress might change, as assets for the game are downloaded. When this happens, React will re-invoke the `render` method. Let us say that the progress changes from 0% to 20%. Before the change, React knows that the last component descriptor it mounted underneath `App` was:

```
<ProgressBar value=0 />
```

when the progress is changed, `render` returns the component descriptor:

```
<ProgressBar value=20 />
```

At this point, React observes that the two component descriptors are of the same type (`ProgressBar` and `ProgressBar`), but that one of the properties has changed. Since the components are the same, React simply updates the property `value` in the mounted component `ProgressBar` and calls its `render` method. This is called reconciliation.

Now, let us consider what happens when the progress reaches 100%. React knows that the last component descriptor it mounted underneath `App` was:

```
<ProgressBar value=20 />
```

when the progress is changed to 100%, `render` returns the component descriptor:

```
<Game />
```

At this point, React observes that the two component descriptors are *not* of the same type (`ProgressBar` and `Game`), hence it unmounts `ProgressBar` (destroying it) and then it mounts `Game` in its place and calls its `render` method.

In summary, a component descriptor is a value that is a static description of a component that can be turned into a mounted component React. The goal of React is to ensure that whatever component descriptors are returned by `render`, they are kept consistent with the currently mounted components, and that `render` is invoked whenever a change happens that might change its output.

3.3 Component State and Properties

Each React component has a set of properties and an internal state. The properties are accessed through the `this.props` field inside the component, whereas the state is accessed through `this.state`. Importantly, *neither field should be changed directly!* The properties

of a component are always derived from the properties described in a component descriptor, e.g., `<ProgressBar value=20 />`. To change the state of a component, the programmer must explicitly call `setState` on the component. These two patterns ensure that React always knows when changes occur to the properties or state of a mounted component. If properties or state were to be changed through other means, the component might become out of sync with its visual representation in the virtual DOM (and real DOM).

In React, the data flow of properties and state can be quite complex. In general, the state of a component can be passed as a property to another component. However, it is also possible, to use a property as part of a component's own internal state (e.g., by passing it to `setState`), or to derive state from a property. For example, in the motivating example, the `filter` is a part of the *state* of the `App` component, but it is passed as a *property* to the `RssFeed` component.

3.4 Render and Child Components

The `render` method is at the heart of each React component. It determines the subcomponents of a component by returning component descriptors. A small, but important detail is that it only determines *one level* of components. For an example, consider the following program:

```

72 class Component extends React.Component {
73     render() {
74         return (
75             <Subcomponent>
76                 <Button>Click Me</Button>
77             </Subcomponent>
78         );
79     }
80 }
81 class Subcomponent extends React.Component {
82     render() {
83         return (<h1>Hello World!</h1>);
84     }
85 }

```

Here, the `render` method of the `Component` class returns a `<Subcomponent>...</Subcomponent>` descriptor with a `Button` component descriptor inside it. Our intuition tells us that `Subcomponent` should have a button somewhere inside it, but this is not necessarily so. In fact, there is no guarantee that the `Button` component descriptor is ever mounted. To understand why, consider the `render` method of the `Subcomponent`. This method unconditionally returns an `h1` tag. Hence if we were to render `Component` all we would see would be a `h1` tag. In React, nested component descriptors are simply treated as a special property called `children`. A component must explicitly refer to this property to use any component descriptors that may be nested within it. For example, we could change the subcomponent to:

```

86 class Subcomponent extends React.Component {
87     render() {
88         return (
89             <div>
90                 <h1>Goodbye World!</h1>
91                 {this.props.children}
92             </div>
93         );
94     }
95 }

```

In this case, the subcomponent would also mount the `Button` component descriptor passed by its parent component. Hence, in the semantics we will write component descriptors as `<C props... />` ignoring any nested component descriptors, as these are simply passed as a special property called `children`.

12:10 A Semantics for the Essence of React

$c \in Cst$	$=$	$bool \mid num \mid str \mid null \mid undef$	[constant]
$v \in Val$	$=$	c	[literal]
		$ a$	[address]
		$ \{str : v \dots\}$	[object]
		$ \lambda(x \dots) e$	[function]
$e \in Exp$	$=$	v	[value]
		$ x$	[variable]
		$ e ; e$	[sequence]
		$ e = e$	[assignment]
		$ \mathbf{let} (x = e) e$	[binding]
		$ e(e \dots)$	[call]
		$ e.f$	[field load]
		$ e.f = e$	[field store]
		$ \mathbf{ref} e$	[address of]
		$ \mathbf{deref} e$	[value at]
$x \in Var$	$=$	is a finite set of variable names.	
$f \in Fld$	$=$	is a finite set of field names.	
$a \in Addr$	$=$	is an infinite set of memory addresses.	
$\lambda \in Lam$	$=$	is the set of all lambda expressions.	

■ **Figure 3** Syntax of λ_{js} .

3.5 Syntax of λ_{react}

We are now ready to present the syntax of λ_{react} . We assume a base language with support for objects and references such as λ_{js} [10], shown in Figure 3.

Values

We extend the values of λ_{js} with two new central concepts: *component descriptors* and *mounted components*. Figure 4 and Figure 5 show the extended grammar of values in λ_{react} . A component descriptor is written as $\langle C \ props \rangle$ where C is an identifier and $props$ is an object literal, i.e., a set of key-value pairs. In React, C is a class, but for our purposes it is sufficient that C is an identifier. A mounted component is written as $\langle C@a \ props \rangle$ and is similar to a component descriptor, except that it is associated with an object in the heap stored at address a . A component descriptor is just that; a “dead” description, whereas a mounted component is a “live” object. We will write π to refer to component descriptors and Π to refer to mounted components. The mnemonic is that mounting a component descriptor changes it from π to Π .

Expressions

We extend the syntax of λ_{js} with React constructs for mounting, unmounting, and reconciliation of components. Figure 6 shows the grammar of the new constructs. We briefly explain each new expression; their semantics are explained in-depth in the following subsection.

$$\begin{aligned}
\pi \in \text{Component Descriptor} &= \langle C \text{ props} \rangle \\
\Pi \in \text{Mounted Component} &= \langle C @ a \text{ props} \rangle \\
\text{props} &= \{k_1 = v_1, \dots, k_n = v_n\} \\
C &= \text{is a set of identifiers.}
\end{aligned}$$

■ **Figure 4** Component Descriptors and Mounted Components.

$$v \in \text{Val} = \dots \mid \pi \mid \Pi$$

■ **Figure 5** Values of λ_{react} .

$\text{MOUNT}(e)$ is used to mount a component descriptor. $\text{UNMOUNT}(e)$ is used to unmount a mounted component. $\text{MOUNTSEQ}(e)$ and $\text{UNMOUNTSEQ}(e)$ are variants of these that operate on sequences of component descriptors and mounted components, respectively. $\text{MOUNTED}(e)$ and $\text{UNMOUNTED}(e)$ are used to perform cleanup after a mount or unmount operation has completed. $\text{RECONCILE}(e, e)$ is used to reconcile a component descriptor with a mounted component. Reconciliation, as will be explained, is the process of updating a mounted component with new data; either through an incremental re-render of the affected subcomponents or through unmounting/mounting. $\text{RECONCILESEQ}(e, e)$ is similar, but reconciles a sequence of component descriptors with a sequence of mounted components. Finally, the \bullet expression represents the event-loop, which marks when an event listener can be executed.

Evaluation Context

We extend λ_{react} with the evaluation contexts shown in Figure 7.

Notation

We will write \bar{a} for a sequence of addresses, $\bar{\pi}$ for a sequence of component descriptors, and $\bar{\Pi}$ for a sequence of mounted components. We will write the empty sequence as *Nil*. We use pattern matching $\pi :: \bar{\pi}$ to deconstruct a sequence into its head (π) and its tail ($\bar{\pi}$). Given a partial map $f : A \leftrightarrow B$, we write $f - a$ for the same map, but with the binding for a removed. We write the empty map as \emptyset .

$$\begin{aligned}
e \in \text{Exp} &= \dots \\
& \mid \text{SetState}(e, e) \mid \text{RENDER}(e) \mid \text{RERENDER}(e) \\
& \mid \text{MOUNT}(e) \mid \text{MOUNTSEQ}(e) \mid \text{MOUNTED}(e, e) \\
& \mid \text{UNMOUNT}(e) \mid \text{UNMOUNTSEQ}(e) \mid \text{UNMOUNTED}(e) \\
& \mid \text{RECONCILE}(e, e) \mid \text{RECONCILESEQ}(e, e) \\
& \mid \bullet
\end{aligned}$$

■ **Figure 6** Syntax of λ_{react} .

12:12 A Semantics for the Essence of React

$$\begin{aligned}
 E &= \square \\
 &| \text{SetState}(E, e) \mid \text{SetState}(v, E) \mid \text{RENDER}(E) \mid \text{RENDER}(E) \\
 &| \text{MOUNT}(E) \mid \text{MOUNTSEQ}(E) \mid \text{MOUNTED}(E, e) \mid \text{MOUNTED}(v, E) \\
 &| \text{UNMOUNT}(E) \mid \text{UNMOUNTSEQ}(E) \mid \text{UNMOUNTED}(E, e) \mid \text{UNMOUNTED}(v, E) \\
 &| \text{RECONCILE}(E, e) \mid \text{RECONCILE}(v, E) \mid \text{RECONCILESEQ}(E, e) \mid \text{RECONCILESEQ}(v, E)
 \end{aligned}$$

■ **Figure 7** Evaluation Contexts for λ_{react} .

$$\begin{aligned}
 \sigma \in \text{Heap} &= \text{Addr} \leftrightarrow \text{Val} \\
 \delta \in \text{ComponentState} &= \text{Addr} \leftrightarrow \text{Obj} \\
 \zeta \in \text{ComponentShape} &= \text{Addr} \leftrightarrow \text{Mounted Component} \times (\text{Address})^* \\
 \ell \in \text{Listeners} &= \text{Addr} \leftrightarrow \mathcal{P}(\text{Lam}) \\
 \chi \in \text{Configuration} &= \text{Heap} \times \text{ComponentState} \times \text{ComponentShape} \times \text{Listeners} \times \text{Exp}
 \end{aligned}$$

■ **Figure 8** Runtime of λ_{react} .

3.6 Runtime of λ_{react}

The runtime of λ_{react} is conceptually similar to λ_{js} , but extended with several additional aspects to keep track of React components. Figure 8 shows the runtime of λ_{react} . A configuration $\chi \in \text{Configuration}$ is a 5-tuple $\langle \sigma, \delta, \zeta, \ell, e \rangle$ consisting of the heap σ , the component state map δ , the component shape map ζ , the listener map ℓ , and an expression e . A heap σ is a partial map from addresses to values. A component state map δ is a partial map from (component) addresses to objects. The component state map does *not* hold the current state of a component, but rather its *next* state which will become the current state through the process of reconciliation. (The current state of a component is always available through the `state` field on the component object.) A component shape map ζ is a partial map from (component) addresses to pairs of a mounted component and a sequence of addresses. The component shape map records the current “shape” of a mounted component along with its currently mounted subcomponents. Intuitively, the component shape map can be thought of as the “Virtual DOM”; what the browser is currently displaying. A listener map ℓ is a partial map from (component) addresses to a set of lambda expressions. The map holds the currently registered event listeners associated with a mounted component. Finally, every configuration has an expression e .

3.7 Initial State

A λ_{react} program consists of a single root component descriptor π (e.g., `<App props />`). We define a function, `inject`, to insert the component descriptor into an empty, initial configuration:

$$\text{inject}(\pi) = \langle \emptyset, \emptyset, \emptyset, \emptyset, \text{MOUNT}(\pi); \bullet \rangle$$

The initial configuration starts with an empty heap ($\sigma = \emptyset$), an empty component state map ($\delta = \emptyset$), an empty component shape map ($\zeta = \emptyset$), and an empty map of listeners ($\ell = \emptyset$). The initial expression is `MOUNT(π); \bullet` which will trigger a mount of the root component descriptor, recursively mounting its subcomponents, and registering event listeners on all mounted components. Once the root component is mounted, the expression enters the event loop, and begins to execute event listeners non-deterministically.

$$\begin{array}{c}
\text{keys}(o_1) = \text{keys}(o_2) \\
\forall k \in \text{keys}(o_1). o_1(k) \text{ is primitive} \Rightarrow o_1(k) == o_2(k) \\
\forall k \in \text{keys}(o_1). o_1(k) \text{ is reference} \Rightarrow o_1(k) === o_2(k) \\
\hline
o_1 \equiv o_2
\end{array}
\quad [\equiv\text{-PROPS}]$$

■ **Figure 9** React Object Equivalence.

$$\begin{array}{c}
\forall k_i \in \text{keys}(o_1). o_1(k_i) = v_i \\
\forall k'_i \in (\text{keys}(o_2) - \text{keys}(o_1)). o_2(k'_i) = v'_i \\
o_3 = \{k_1 : v_1, \dots, k_n : v_n, k'_1 : v'_1, \dots, k'_n : v'_n\} \\
\hline
o_1 \otimes o_2 = o_3
\end{array}
\quad [\otimes\text{-STATE}]$$

■ **Figure 10** The State Merge Operator \otimes .

3.8 Semantics of Object Equality

A key React operation is to determine when two objects are equal. React uses this to determine when property- and state objects are unchanged during reconciliation, as discussed later. Figure 9 defines two objects to be equal if

- (i) they share the same keys,
- (ii) the values of primitive types are compared by equality, and
- (iii) the references are compared using reference equality.

For example,

$$\{a : 21, b : 42\} \equiv \{a : 21, b : 42\}$$

and

$$\{a : 21, b : \ell\} \equiv \{a : 21, b : \ell\}$$

where ℓ is some address in the heap.

This shallow notion of equality can be checked efficiently, since we never have to recursively descend into the object structure.

3.9 Semantics of State Merges

Another key React operation is to merge two objects. Like equivalence, merging is also a shallow operation. Specifically, two objects are merged in a left-biased manner where the returned object is obtained by taking all keys and values from the left object and adding those keys and values from the right object that did not appear in the left object. Figure 10 captures this notion.

For example,

$$\{a : 21, b : 42\} \otimes \{b : 84\} = \{a : 21, b : 42\}$$

and

$$\{a : 21, b : 42\} \otimes \{c : 84\} = \{a : 21, b : 42, c : 84\}$$

$$\begin{array}{c}
\frac{\pi = \langle C \text{ props} / \rangle \quad a \notin \text{dom}(\sigma) \quad \sigma' = \sigma[a \mapsto \{\text{props} : \text{props}\}] \quad \delta' = \delta[a \mapsto \{\}] \quad \ell' = \ell[a \mapsto \text{listenersOf}(\text{props})]}{\langle \sigma, \delta, \zeta, \ell, \text{MOUNT}(\pi) \rangle \rightarrow \langle \sigma', \delta', \zeta, \ell', \text{MOUNTED}(\langle C @ a \text{ props} / \rangle, \text{MOUNTSEQ}(\text{RENDER}(\pi))) \rangle} \quad (\text{E-MOUNT}) \\
\\
\frac{\zeta' = \zeta[a \mapsto (\langle C @ a \text{ props} / \rangle, \bar{a})]}{\langle \sigma, \delta, \zeta, \ell, \text{MOUNTED}(\langle C @ a \text{ props} / \rangle, \bar{a}) \rangle \rightarrow \langle \sigma, \delta, \zeta', \ell, a \rangle} \quad (\text{E-MOUNTED}) \\
\\
\frac{}{\langle \sigma, \delta, \zeta, \ell, \text{MOUNTSEQ}(\text{Nil}) \rangle \rightarrow \langle \sigma, \delta, \zeta, \ell, \text{Nil} \rangle} \quad (\text{E-MOUNT-SEQ-1}) \\
\\
\frac{}{\langle \sigma, \delta, \zeta, \ell, \text{MOUNTSEQ}(\pi :: \bar{\pi}) \rangle \rightarrow \langle \sigma, \delta, \zeta, \ell, \text{MOUNT}(\pi) :: \text{MOUNTSEQ}(\bar{\pi}) \rangle} \quad (\text{E-MOUNT-SEQ-2}) \\
\\
\frac{\zeta(a) = (\langle C @ a \text{ props} / \rangle, \bar{a})}{\langle \sigma, \delta, \zeta, \ell, \text{UNMOUNT}(a) \rangle \rightarrow \langle \sigma, \delta, \zeta, \ell, \text{UNMOUNTSEQ}(\bar{a}); \text{UNMOUNTED}(a) \rangle} \quad (\text{E-UNMOUNT}) \\
\\
\frac{\ell' = \ell - a}{\langle \sigma, \delta, \zeta, \ell, \text{UNMOUNTED}(a) \rangle \rightarrow \langle \sigma, \delta, \zeta, \ell', \text{Nil} \rangle} \quad (\text{E-UNMOUNTED}) \\
\\
\frac{}{\langle \sigma, \delta, \zeta, \ell, \text{UNMOUNTSEQ}(\text{Nil}) \rangle \rightarrow \langle \sigma, \delta, \zeta, \ell, \text{Nil} \rangle} \quad (\text{E-UNMOUNT-SEQ-1}) \\
\\
\frac{}{\langle \sigma, \delta, \zeta, \ell, \text{UNMOUNTSEQ}(a :: \bar{a}) \rangle \rightarrow \langle \sigma, \delta, \zeta, \ell, \text{UNMOUNT}(a); \text{UNMOUNTSEQ}(\bar{a}) \rangle} \quad (\text{E-UNMOUNT-SEQ-2})
\end{array}$$

■ **Figure 11** Semantics of mounting and unmounting components.

Note that the procedure is *not* recursive:

$$\{o : \{a : 21\}\} \otimes \{o : \{b : 42\}\} = \{o : \{a : 21\}\}$$

which is common source of bugs.

Both object equality and merging play vital roles in the reconciliation of components.

3.10 Semantics of Mounting and Unmounting

We now discuss the process of mounting and unmounting components. A component is mounted when a λ_{react} application starts and it may cause subcomponents to be mounted, and so on recursively. Components are also mounted and unmounted as part of the reconciliation process, which will be described later. Figure 11 shows the semantics of mounting and unmounting components. We now discuss each evaluation rule in greater detail:

[E-Mount]

The rule states that to mount a component descriptor $\pi = \langle C \text{ props} / \rangle$ the following steps are taken: A fresh address $a \notin \text{dom}(\sigma)$ is chosen. An object is stored at that address in the heap with a copy of the *props* object. The component state map δ is updated with a binding for a , binding it to the empty object (since the next pending state is currently empty). The event listeners are extracted from the *props* object and registered in the event listener map ℓ . These steps are sufficient to mount the component, but we must also recursively mount its subcomponents as determined by its **render** method.

We achieve this by having the mount expression reduce to:

$$\text{MOUNTED}(\langle C @ a \text{ props} / \rangle, \text{MOUNTSEQ}(\text{RENDER}(\pi)))$$

which can be understood as follows: The inner $\text{RENDER}(\pi)$ will reduce to a sequence of subcomponent descriptors. We will then mount each of these in turn. This will reduce to

a sequence of mounted component addresses \bar{a} . Finally, the $\text{MOUNTED}(\langle C@a \text{ props}/\rangle, \bar{a})$ expression will register that the mounted components addresses \bar{a} are subcomponents of the current component a .

[E-Mounted]

The rule states that the expression $\text{MOUNTED}(\langle C@a \text{ props}/\rangle, \bar{a})$ reduces to the address a of the mounted component $\langle C@a \text{ props}/\rangle$ with the component shape map ζ updated to reflect the current shape of the component a and that \bar{a} are the current subcomponents of a . Intuitively, once the `Mounted` expression is evaluated, the component a and its subcomponents have been fully mounted, and we have recorded their shape so that, in the future when we re-render the component, we are able to compare the current shape to the component descriptors returned by `render`.

[E-Mount-Seq-1]

The rule states that mounting the empty sequence of component descriptors results in the empty sequence of mounted component addresses.

[E-Mount-Seq-2]

The rule states that to mount a sequence of component descriptors $\pi :: \bar{\pi}$ we mount the first component π and then we mount the remaining component descriptors $\bar{\pi}$. Mounting a component descriptor π returns a mounted component address a and since our goal is to produce a sequence of mounted component addresses, we prepend the result of mounting π with the result of mounting the remaining component descriptors $\bar{\pi}$. Thus, `MountSeq` always reduces to a sequence of mounted component addresses.

For example, if we mount the two component descriptors:

$$\langle \sigma, \delta, \zeta, \ell, \text{MOUNTSEQ}(\langle \text{TextField props1}/\rangle :: \langle \text{Button props2}/\rangle :: \text{Nil}) \rangle$$

we obtain a new configuration with the two mounted components:

$$\langle \sigma', \delta', \zeta', \ell', a_1 :: a_2 :: \text{Nil} \rangle$$

where

$$\zeta'(a_1) = \langle \text{TextField}@_{a_1} \text{ props1}/\rangle \quad \text{and} \quad \zeta'(a_2) = \langle \text{Button}@_{a_2} \text{ props2}/\rangle$$

[E-Unmount]

The rule states that to unmount a mounted component address a , we must first unmount its subcomponents, which are known from the component shape map ζ , and afterwards we can consider the component Π to be unmounted.

[E-Unmounted]

The rule states that once the subcomponents of a component a have been unmounted, all listeners are removed from the event listener map ℓ . We do *not* remove the address a from the heap σ since there could still be a reference to the component object somewhere nor do we remove it from the component shape map ζ . A garbage collector can be used to clean these maps, if desired.

$$\begin{array}{c}
\frac{}{\langle \sigma, \delta, \zeta, \ell, \text{RECONCILESEQ}(\text{Nil}, \text{Nil}) \rangle \rightarrow \langle \sigma, \delta, \zeta, \ell, \text{Nil} \rangle} \text{(RC-EMPTY)} \\
\frac{}{\langle \sigma, \delta, \zeta, \ell, \text{RECONCILESEQ}(\bar{\pi}, \text{Nil}) \rangle \rightarrow \langle \sigma, \delta, \zeta, \ell, \text{MOUNTSEQ}(\bar{\pi}) \rangle} \text{(RC-EXTEND)} \\
\frac{}{\langle \sigma, \delta, \zeta, \ell, \text{RECONCILESEQ}(\text{Nil}, \bar{a}) \rangle \rightarrow \langle \sigma, \delta, \zeta, \ell, \text{UNMOUNTSEQ}(\bar{a}) \rangle} \text{(RC-TRUNCATE)} \\
\frac{}{\langle \sigma, \delta, \zeta, \ell, \text{RECONCILESEQ}(\pi :: \bar{\pi}, a :: \bar{a}) \rangle \rightarrow \langle \sigma, \delta, \zeta, \ell, \text{RECONCILE}(\pi, a) :: \text{RECONCILESEQ}(\bar{\pi}, \bar{a}) \rangle} \text{(RC-SEQUENCE)} \\
\frac{\pi = \langle C_1 \text{ nextProps} / \rangle \quad \zeta(a) = (\langle C_2 @ a \text{ prevProps} / \rangle, _) \quad C_1 \neq C_2}{\langle \sigma, \delta, \zeta, \ell, \text{RECONCILE}(\pi, a) \rangle \rightarrow \langle \sigma, \delta, \zeta, \ell, \text{UNMOUNT}(a) ; \text{MOUNT}(\pi) \rangle} \text{(RC-DIFF-ROOT)} \\
\frac{\pi = \langle C \text{ nextProps} / \rangle \quad \zeta(a) = (\langle C @ a \text{ prevProps} / \rangle, \bar{a}) \quad \text{nextState} = \delta(a) \\ o = \sigma(a) \quad o' = o[\text{props} \mapsto \text{nextProps}][\text{state} \mapsto \text{nextState}] \quad \sigma' = \sigma[a \mapsto o']}{\langle \sigma, \delta, \zeta, \ell, \text{RECONCILE}(\pi, a) \rangle \rightarrow \langle \sigma', \delta, \zeta, \ell, \text{RECONCILED}(\langle C @ a \text{ nextProps} / \rangle, \text{RECONCILESEQ}(\text{RENDER}(a), \bar{a})) \rangle} \text{(RC-SAME-ROOT)} \\
\frac{\zeta' = \zeta[a \mapsto (\langle C @ a \text{ props} / \rangle, \bar{a})]}{\langle \sigma, \delta, \zeta, \ell, \text{RECONCILED}(\langle C @ a \text{ props} / \rangle, \bar{a}) \rangle \rightarrow \langle \sigma, \delta, \zeta', \ell, a \rangle} \text{(RC-RECONCILED)}
\end{array}$$

■ **Figure 12** Semantics of reconciliation.

[E-Unmount-Seq-1]

The rule states that unmounting the empty sequence of mounted component addresses results in the empty sequence.

[E-Unmount-Seq-2]

The rule states that to unmount a sequence of mounted component addresses $a :: \bar{a}$ we must unmount the first component a and then we can unmount the remaining components \bar{a} .

It is easy to see how the structure of the [E-MOUNT], [E-MOUNTED], [E-MOUNT-SEQ-1], and [E-MOUNT-SEQ-2] mirror the structure of [E-UNMOUNT], [E-UNMOUNTED], [E-UNMOUNT-SEQ-1], and [E-UNMOUNT-SEQ-2]. Mounting is essentially a recursive traversal of a tree that is gradually being computed by the `render` methods. Unmounting is the reverse process, using the component shape map to recursively remove subcomponents.

3.11 Semantics of Reconciliation

The purpose of reconciliation is to merge a component descriptor (or sequence of component descriptors) with a mounted component (or sequence of mounted components). At a high level, there are two broad cases to consider: (i) a mounted component is updated with new properties and state, and (ii) a mounted component is replaced by another component. We introduce two expressions: $\text{RECONCILE}(e, e)$ and $\text{RECONCILESEQ}(e, e)$ to model reconciliation. The former reconciles a component descriptor with a mounted component address, whereas the latter deals with sequences of component descriptors and mounted component addresses. Figure 12 shows the semantics of reconciliation. We now discuss each rule in greater detail:

[RC-Empty]

The rule states that reconciliation of the empty sequence of component descriptors with the empty sequence of mounted component addresses simply results in the empty sequence of mounted component addresses.

[RC-Extend]

The rule states that reconciliation of a sequence of component descriptors $\bar{\pi}$ with the empty sequence of mounted component addresses Nil results in each of the $\bar{\pi}$ component descriptors being mounted. For example, if we were to reconcile the component descriptors:

```
<RssFeed title="..." /> :: <RssFeed title="..." />
```

with the empty sequence of mounted component addresses then we would simply mount the two `<RssFeed>`s. This is a common occurrence when the `render` method of a component returns additional component descriptors. The rule is called *extend* because it *extends* a sequence of subcomponents with additional components.

[RC-Truncate]

The rule states that reconciliation of the empty sequence of component descriptors with a sequence of \bar{a} of mounted component addresses results in each of the \bar{a} components being unmounted. For example, if we were to reconcile the empty sequence of component descriptors with the sequence of mounted component addresses:

$$a_1 :: a_2 :: Nil$$

where

$$\zeta(a_1) = \langle a1@RssFeed \text{ title}="..." /> \quad \zeta(a_2) = \langle a2@RssFeed \text{ title}="..." />$$

then the mounted components a_1 and a_2 would be unmounted. The rule is called *truncate* because it *truncates* a sequence of subcomponents. Intuitively, this is the dual of the [RC-EXTEND] rule.

[RC-Sequence]

The rule states that reconciliation of a sequence of component descriptors $\pi :: \bar{\pi}$ with a sequence of mounted component addresses $a :: \bar{a}$ requires pairwise reconciliation, i.e., we have to reconcile π with a and then reconcile the rest of the two sequences $\bar{\pi}$ and \bar{a} . For example, if we were to reconcile the sequence of component descriptors:

```
<RssFeed title="The Guardian" /> :: <RssFeed title="Reuters" />
```

with the sequence of mounted component addresses:

$$a_1 :: a_2 :: Nil$$

where

$$\zeta(a_1) = \langle a1@RssFeed \text{ title}="BBC World" /> \quad \zeta(a_2) = \langle a2@RssFeed \text{ title}="Reuters" />$$

the first component descriptor would be reconciled with the first mounted component a_1 and similarly the second component descriptor would be reconciled with the second mounted component a_2 . In this case, the first component a_1 would be re-rendered and recursively reconciled since one of its properties changed.

We now turn to the more interesting question of how to reconcile a single component descriptor with a single mounted component. As stated previously, there are two cases to consider: (i) a mounted component is being updated with new properties and state, or (ii) a mounted component is being replaced by another component. We begin with the latter.

[RC-Diff-Root]

The rule states that reconciliation of a component descriptor $\pi = \langle C_1 \text{ nextProps} \rangle$ with a mounted component address a where $\zeta(a) = \langle a@C_2 \text{ prevProps} \rangle$ and where the descriptor and the mounted components have different kinds, i.e., $C_1 \neq C_2$, is a two-step process. First, the currently mounted component C_2 is unmounted, which as we have seen, will recursively unmount its subcomponents. Second, the C_1 component descriptor is mounted.

For example, if we were to reconcile the component descriptor:

```
<Alert color="secondary">Submitted!</Alert>
```

with the mounted component address a where:

```
 $\zeta(a) = \langle a@Button \text{ color="primary">Submit</Button}>$ 
```

The mounted `Button` component would be unmounted, and the `Alert` component descriptor would be mounted in its place. One component being replaced by another component is a common occurrence in the implementation of form dialogs and page navigation.

[RC-Same-Root]

This rule states that reconciliation of a component descriptor $\pi = \langle C \text{ nextProps} \rangle$ with a mounted component address a where $\zeta(a) = \langle C@a \text{ prevProps} \rangle$ and the descriptor and mounted component have the same kind requires multiple steps: The `props` field of the component object is updated to `nextProps`. Similarly, the `state` field is updated to the value of the next state as specified by the component state map δ . Finally, the expression reduces to the expression:

$$\text{RECONCILED}(\Pi, \text{RECONCILESEQ}(\text{RENDER}(a), \bar{a}))$$

since we must re-render the component and reconcile the returned component descriptors (which could have changed) with the currently mounted subcomponents \bar{a} . Once this is done, we must update the component shape map of a to store its newly mounted / reconciled subcomponents, hence we wrap the result in `RECONCILED` which is similar to `MOUNTED` and `UNMOUNTED`.

A variant of the [RC-SAME-ROOT] rule, closer to React semantics, would use React's object equivalence to determine whether the `prevProps` and `nextProps` are equivalent, and whether `nextState` and the current state are equivalent. If all were found to be equivalent then there is no need to do anything, and we could simply skip the updates and re-rendering. This is a performance consideration, hence we have omitted it from the rules.

[Rc-Reconciled]

The rule states that once reconciliation is complete for the mounted component address a with (possibly new) subcomponents \bar{a} then we update the component shape map ζ to store the subcomponents and then return the component address a itself.

3.12 Semantics of Rendering

The semantics of rendering, shown in Figure 13, are straightforward. As mentioned earlier, there are two types of rendering: rendering a component descriptor for the first time and re-rendering an already mounted component.

$$\begin{array}{c}
\frac{\pi = \langle C \text{ props } / \rangle \quad \langle \sigma, \delta, \zeta, \ell, \mathbf{props.render}() \rangle \rightarrow \langle \sigma, \delta, \zeta, \ell, \bar{\pi} \rangle}{\langle \sigma, \delta, \zeta, \ell, \mathbf{RENDER}(\pi) \rangle \rightarrow \langle \sigma, \delta, \zeta, \ell, \bar{\pi} \rangle} \quad \text{[E-RENDER]} \\
\frac{\zeta(a) = (\langle C@a \text{ props } / \rangle, _) \quad \langle \sigma, \delta, \zeta, \ell, \mathbf{props.render}() \rangle \rightarrow \langle \sigma, \delta, \zeta, \ell, \bar{\pi} \rangle}{\langle \sigma, \delta, \zeta, \ell, \mathbf{RENDER}(a) \rangle \rightarrow \langle \sigma, \delta, \zeta, \ell, \bar{\pi} \rangle} \quad \text{[E-RENDER]}
\end{array}$$

■ **Figure 13** Semantics of Rendering and Re-Rendering.

$$\frac{\text{nextState} = \delta(a) \quad \delta' = \delta[a \mapsto \text{newState} \otimes \text{nextState}] \quad \zeta(a) = (\langle C@a \text{ props } / \rangle, \bar{a})}{\langle \sigma, \delta, \zeta, \ell, \mathbf{SetState}(a, \text{newState}) \rangle \rightarrow \langle \sigma, \delta', \zeta, \ell, \mathbf{RECONCILED}(a, \mathbf{RECONCILESEQ}(\mathbf{RENDER}(a), \bar{a})) \rangle} \quad \text{[E-SET-STATE]}$$

■ **Figure 14** Semantics of Set-State.

[E-Render] and [E-ReRender]

The [E-RENDER] rule states that to render a component descriptor we invoke the `render` method of the `props` object. We assume that such a method always exists on `props`. The [E-RENDER] is similar but for a mounted component where we use the address a to retrieve the component from the component shape map ζ and then we call its `render` method of its `props` object.

An interesting observation about [E-RENDER] and [E-RENDER] is that, once a component has been mounted, overwriting its `props.render` will not have any effect, since the `props` object itself is stored in the component shape map ζ . While this may seem overly complicated (and to some extent it is), it (i) is consistent with actual React semantics, and (ii) it allows us to prove key properties of λ_{react} . Specifically, in these proofs, we need to know that the `render` method is not suddenly changed underneath us. Note that calling `render` by itself has no effect in our semantics; it is only when it is called from within e.g., `MOUNTSEQ` and `RECONCILESEQ` that mounting or reconciliation is triggered.

As mentioned earlier, the `render` method must return a sequence of component descriptors. Each component descriptor carries its own properties with a `render` method inside it. For this process to terminate, at some point a component will not have *any* subcomponents and simply return the empty sequence of component descriptors.

3.13 Semantics of State Changes

The semantics of state changes, shown in Figure 14, are also straightforward:

[E-Set-State]

The rule states that if the mounted component a is passed an object `newState` with some new state then we must retrieve the `nextState` from the δ map and merge the current next state with the new state. Finally, we must trigger a reconciliation wrapped in a `RECONCILED` since changing the state of a component could change what is returned by its `render`.

3.14 Semantics of Events

As mentioned earlier, the properties of a component descriptor may contain fields that correspond to various event listeners. For example, if there is a field `onClick` then it should be registered as an event listener when the component descriptor is mounted (and unregistered

$$\frac{a \in \text{Addr} \quad \lambda \in \ell(a)}{\langle \sigma, \delta, \zeta, \ell, \bullet \rangle \rightarrow \langle \sigma, \delta, \zeta, \ell, \lambda(a); \bullet \rangle} \quad [\text{E-LOOP}]$$

■ **Figure 15** Semantics of Events.

when the component is unmounted). In a real React application, such event listeners are executed in response to user events. In the λ_{react} semantics, we add a rule, shown in Figure 15, which states that once we are in the event loop \bullet then we may select any (component) address a and pick any of its event listeners $\lambda \in \ell(a)$, execute it, and then return to the event loop.

In the semantics, as well as in real React applications, execution of an event listener may invoke `setState` which in turn may cause a component to re-render and trigger the process of reconciliation. Thus, at a high-level, the execution of a React application can be understood as an initial mount (as defined by the `inject` function) followed by a sequence of reconciliations caused by calls to `setState` from event listeners.

4 Properties of λ_{react}

We want to show that mounting, unmounting, and reconciliation terminate. However, in general, these processes may not terminate if the user-defined `render` function is badly behaved. Trivially, if `render` does not terminate then mounting a component descriptor will not terminate. But even if we assume that `render` terminates, it could return a list of “recursive” component descriptors. That is, the `render` function of a component descriptor $\langle C \text{ props} / \rangle$ could return a list that includes C itself. This would cause an execution where an infinite tree of component descriptors is mounted (which obviously never terminates).

To overcome these issues, we define the notion of a *well-behaved* component. Simply put, the `render` function of a well-behaved component must always return a list of component descriptors where each component descriptor is strictly “smaller” than the component itself. Under the assumption that components are well-behaved, we can prove properties about mounting, unmounting, and reconciliation.

We now formalize the notions of rank and well-behavedness:

4.1 Definitions

► **Definition 1** (Rank). *A ranking function $\text{rank} : \text{Identifier} \rightarrow \text{Nat}$ is a map from identifiers (component names) to natural numbers.*

► **Definition 2** (k -Well-Behaved Expressions). *An expression e is k well-behaved if it evaluates to a list of component descriptors $\bar{\pi}$ such that for each component descriptor $\pi_i = \langle C \text{ props} / \rangle$ in the list it is the case that $\text{rank}(C) < k$. If $k = 0$ then e must evaluate to the empty list.*

► **Definition 3** (k -Well-Behaved Component Descriptors). *A component descriptor $\pi = \langle C \text{ props} / \rangle$ is k well-behaved if $\text{rank}(C) = k$ and the render function `props.render` is k well-behaved.*

That is to say, the render function can only return component descriptors with a strictly lower rank than the rank of the component.

► **Definition 4** (*k*-Well-Behaved Mounted Components). *A mounted component $\Pi = \langle C @ a \text{ props} \rangle$ is k well-behaved if $\text{rank}(C) = k$ and the render function props.render is k well-behaved.*

That is to say, the `render` function can only return component descriptors with a strictly lower rank than the rank of the mounted component.

► **Definition 5** (Well-Behaved Component Shape Maps). *A component shape map ζ is well-behaved if:*

- *For every $a \in \text{dom}(\zeta)$ where $\zeta(a) = (\Pi, \bar{a})$, Π is k well-behaved for some k and for every address $a_i \in \bar{a}$, $\zeta(a_i) = (\Pi', _)$, Π' is k' well-behaved for some k' where $k' < k$. That is to say, every mounted component is well-behaved, its children are well-behaved, and they have strictly lower rank.*
- *For every pair of addresses a_1 and a_2 with $a_1 \neq a_2$ it is the case that if $\zeta(a_1) = (_, \bar{a}_1)$ and $\zeta(a_2) = (_, \bar{a}_2)$ then the two lists \bar{a}_1 and \bar{a}_2 have disjoint elements. That is to say, every mounted component has exactly one parent.*

As before, if $k = 0$ then the children \bar{a} of a mounted component must be the empty list.

4.2 Theorems

We can now state the main theoretical results of the paper.

► **Theorem 6** (Mount Preserves Well-Behavedness). *If π is a k well-behaved component descriptor and ζ is a well-behaved component shape map then:*

$$\langle \sigma, \delta, \zeta, \ell, E[\text{MOUNT}(\pi)] \rangle \rightarrow^* \langle \sigma', \delta', \zeta', \ell', E[a] \rangle$$

and:

- *ζ' is a well-behaved component shape map,*
- *$\zeta'(a)$ is k well-behaved mounted component, and*
- *a is not the child of any mounted component, i.e. there does not exist an address a_2 such that $\zeta(a_2) = (_, \bar{a}_2)$ where $a \in \bar{a}_2$.*

► **Corollary 7** (Inject is Well-Behaved). *If π is a k well-behaved component then:*

$$\text{inject}(\pi) \rightarrow^* \langle \sigma, \delta, \zeta, \ell, \bullet \rangle$$

where ζ is well-behaved.

► **Theorem 8** (Unmount Preserves Well-Behavedness). *If a is an address in $\text{dom}(\zeta)$ and ζ is a well-behaved component shape map then:*

$$\langle \sigma, \delta, \zeta, \ell, E[\text{UNMOUNT}(a)] \rangle \rightarrow^* \langle \sigma, \delta, \zeta, \ell', E[\text{Nil}] \rangle$$

► **Theorem 9** (Reconciliation Preserves Well-Behavedness). *If π is a k well-behaved component descriptor, ζ is a well-behaved component shape map, $a \in \text{dom}(\zeta)$, $\zeta(a) = (\Pi, _)$, Π is k' well-behaved then*

$$\langle \sigma, \delta, \zeta, \ell, E[\text{RECONCILE}(\pi, a)] \rangle \rightarrow^* \langle \sigma', \delta', \zeta', \ell', E[a'] \rangle$$

and ζ' is well-behaved and $\zeta'(a')$ is k well-behaved.

■ **Table 1** Summary of React Lifecycle Hooks. (★ only under certain conditions.)

Lifecycle Hook	Use <code>setState</code> ?	Deprecated?
<code>constructor(props)</code>	NO	NO
<code>componentDidMount()</code>	YES	NO
<code>componentDidUpdate(prevProps, prevState, snapshot)</code>	YES★	NO
<code>componentWillReceiveProps(nextProps)</code>	YES	YES
<code>componentWillMount()</code>	YES	YES
<code>componentWillUnmount()</code>	NO	NO
<code>componentWillUpdate(nextProps, prevState)</code>	NO	YES
<code>shouldComponentUpdate()</code>	-	NO
<code>getDerivedStateFromProps()</code>	n/a	NO
<code>getSnapshotBeforeUpdate(prevProps, prevState)</code>	-	NO

► **Lemma 10** (ReconcileSeq Preserves Well-Behavedness). *If ζ is a well-behaved component shape map, $\bar{\pi} = \pi_1, \dots, \pi_n$, each π_i is k_i well-behaved, $\bar{a} = a_1, \dots, a_m$, and each $a_i \in \text{dom}(\zeta)$ then*

$$\langle \sigma, \delta, \zeta, \ell, E[\text{RECONCILESEQ}(\bar{\pi}, \bar{a})] \rangle \rightarrow^* \langle \sigma', \delta', \zeta', \ell', E[a'_1, \dots, a'_n] \rangle$$

and ζ' is well-behaved and every $\zeta'(a'_i)$ is k_i well-behaved.

The detailed proofs of these properties are available in a separate technical report [15].

In summary, we have proved that as long as the `render` functions terminate and the component descriptors form a hierarchy that rules out infinite component trees, then the processes of mounting, unmounting, and reconciling components all terminate. The theorems show that these restrictions expressed in terms of React programs are reflected in the runtime state of these programs, and are preserved in the runtime state by all the operations. The theorems also show that these restrictions are sufficient to ensure termination of each of the operations that manipulate components.

5 Lifecycle Hooks

Lifecycle hooks are an important part of React. A lifecycle hook is a callback executed by React in response to changes to a component's properties and state, and when it is mounted or unmounted. Lifecycle hooks are frequently used to acquire (and release) resources, to retrieve data over the internet, and so on. Table 1 shows an overview of the lifecycle hooks available in React. As the figure shows, the design of lifecycle hooks has gone through several iterations, and some lifecycle hooks are now deprecated. Another important aspect of lifecycle hooks is whether they are allowed to call `setState`. This turns out to be quite tricky, because it is easy to accidentally construct infinite loops where a lifecycle hook calls `setState` which in turn triggers a lifecycle hook, and so on. This is source of bugs in React.

The semantics of λ_{react} does not include lifecycles, but we can extend it to accommodate them. For example:

- The `componentWillMount()` method is invoked immediately *before* a component is mounted. In the semantics, this corresponds to the [E-MOUNT] rule, which we could update to trigger a call to `componentWillMount()`.

- The `componentDidMount()` method is invoked immediately *after* a component has been mounted. In the semantics, this corresponds to the [E-MOUNTED] rule, which we could update to trigger a call to `componentDidMount()` immediately before returning the mounted component.
- The `componentWillUnmount()` method is invoked immediately *before* a component is unmounted. In the semantics, this corresponds to the [E-UNMOUNT] rule, which we could update to trigger a call to `componentWillUnmount()`. There is no corresponding `componentDidUnmount()` because changes should not be made to an unmounted component.
- The `componentDidUpdate()` method is invoked immediately *after* a component has been updated. In the semantics, this corresponds approximately to the [RC-SAME-ROOT] reconciliation rule, which we could update to trigger a call to `componentDidUpdate()`.

Note that many of the lifecycle hooks receive the previous properties, the previous state, the new properties, and/or the new state. Since the λ_{react} semantics meticulously models properties and state, these objects are readily available.

6 Related Work

We are not aware of any prior work on formally defining the semantics of React. In this related work section, we focus on previous research on formally specifying the semantics of JavaScript, and on related frameworks for defining user-interfaces declaratively.

Semantics of JavaScript

Many proposals have been made for a formal semantics of JavaScript. Herman and Flanagan [11] presented an implementation of an interpreter for EcmaScript 4 written in ML. Being an interpreter, the specification was executable. However, EcmaScript 4 was never adopted as a standard. Maffei et al. [17] presented the first small-step operational semantics for JavaScript as a basis for formalization of security properties in web applications.

Guha et al. [10] presented λ_{js} , a minimal semantics for JavaScript. A key aspect of their work is to formalize a semantics that is as small as possible, while still being expressive enough to allow compilation of all JavaScript constructs into it. In this way, λ_{js} supports all ugly features of JavaScript, such as prototype-based inheritance, dynamic property access, and implicit coercions.

Gardner et al. [9] presented a program logic based on separation logic for reasoning about a large subset of the ECMAScript 3 language. The subset under consideration includes features such as prototype inheritance and the `with` construct, which interacts with JavaScript's scoping rules in intricate ways.

Park et al. [21] presented KJS, a complete formalization of ECMAScript 5.1 implemented in the K Framework [21]. Being specified in the K framework, the semantics is executable and has been tested against all 2,782 tests in the ECMAScript 5.1 conformance test suite. By specifying all of JavaScript, and executing all test cases, the authors were able to find evaluation rules not covered by any existing test, add tests for these rules, and then running them on different browsers, which ultimately revealed several implementation bugs.

Bodin et al. [6], presented JSCert, a formal semantics for the ECMAScript 5 version of JavaScript that is formalized and proven correct using the Coq proof assistant. Their work also includes a reference interpreter, JSRef, that can be used to execute test cases and compare results against standard JavaScript interpreters. As is typical in formalizations, JSCert excludes a number of pragmatic details such as certain native library functions, and

relies on an external parser to implement `eval`. Also, the `for-in` construct has not been formalized because the standard defines it very loosely. Bodin’s dissertation [7] explored the challenges associated with the formalization in greater detail.

The semantics of asynchronous JavaScript has been tackled by several authors. Madsen et al. [16] proposed an extension of λ_{js} that models events, event listeners, and the event loop. Based on this semantics, the authors developed a static analysis to discover simple bugs in event-driven JavaScript programs. Loring et al. [13] and Madsen et al. both [14] proposed semantics to specify the behavior of JavaScript promises. Later work by Alimadadi et al. [3] presented a tool for finding bugs in promise-based JavaScript code based on [14].

Other Frameworks

A discussion about the design of React and how it evolved can be found in CACM [22]. Since the introduction of React, many other frameworks have appeared that emulate its declarative and object-oriented programming style. React Native [12] lets programmers write native mobile applications using JavaScript and React. React Native uses the React model, but with the UI components of the underlying OS, e.g. iOS or Android. Preact [1] is a light-weight “close to the metal” React-style library with a focus on performance. Preact aims to provide the thinnest possible “virtual DOM” on top of the real DOM. Vue.js [2] is another React-like library with a focus on the view-layer and on easy integration with other existing libraries.

We believe our semantics offers a solid foundation for understanding and potentially modeling these React-inspired libraries. We think that the popularity of React and the adoption of the “React model” by many other frameworks is a sign of its importance for the future of web development.

7 Conclusions and Future Work

React is a framework that enables programmers to write web applications in a declarative and object-oriented style that facilitates reuse. Each component of a React application has a set of properties representing input parameters, a state consisting of values that may vary over time, and a `render` method that specifies its subcomponents. When state changes occur, React’s reconciliation mechanism determines their impact and updates the user-interface incrementally by mounting, unmounting, or reconciling subcomponents selectively. At designated points in this process, the React framework invokes lifecycle methods that enable programmers to perform actions outside the framework such as acquiring and releasing resources. Since these mechanisms exhibit considerable complexity, programmers would benefit from program analyses and tools that can reason precisely about React programs.

It is our long-term goal to develop program understanding and bug finding tools for React applications. To our knowledge, this paper presents the first formal specification of a semantics that captures the essence of React, thus establishing a foundation for such tools. Our small-step operational semantics extends the λ_{js} calculus [10] and models three key concepts of React:

- (i) mounting and unmounting of components,
- (ii) reconciliation of component descriptors and mounted components, and
- (iii) the semantics of state changes.

To demonstrate that key operations such as mounting, unmounting, and reconciliation terminate, we define the notion of a well-behaved component by imposing a ranking function on components, and requiring that the `render` method of a component only returns components of strictly smaller rank. We then prove that well-behavedness is preserved by these operations.

For future work, we plan to conduct a case study to identify and classify common bug patterns in React web applications. Then, with a formal semantics of React in place, we will develop static analysis techniques to detect instances of these bug patterns in React applications. Another avenue for future work is the development of a type system that is sufficiently expressive to capture the lifecycle of React components and ensure that properties and state are not accessed or modified incorrectly.

References

- 1 Preact: Fast 3kB alternative to React with the same modern API. <https://preactjs.com/>, 2019. Accessed: 2019-12-19.
- 2 Vue.js: The progressive JavaScript framework. <https://vuejs.org/>, 2019. Accessed: 2019-12-19.
- 3 Saba Alimadadi, Di Zhong, Magnus Madsen, and Frank Tip. Finding broken promises in asynchronous JavaScript programs. *PACMPL*, 2(OOPSLA):162:1–162:26, 2018. doi:10.1145/3276532.
- 4 Shay Artzi, Adam Kiezun, Julian Dolby, Frank Tip, Danny Dig, Amit M. Paradkar, and Michael D. Ernst. Finding bugs in web applications using dynamic test generation and explicit-state model checking. *IEEE Trans. Software Eng.*, 36(4):474–494, 2010. doi:10.1109/TSE.2010.31.
- 5 SungGyeong Bae, Hyunghun Cho, Inho Lim, and Sukyoung Ryu. SAFEWAPI: Web API misuse detector for web applications. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2014*, pages 507–517, New York, NY, USA, 2014. ACM. doi:10.1145/2635868.2635916.
- 6 Martin Bodin, Arthur Charguéraud, Daniele Filaretti, Philippa Gardner, Sergio Maffeis, Daiva Naudziuniene, Alan Schmitt, and Gareth Smith. A trusted mechanised JavaScript specification. In *The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '14, San Diego, CA, USA, January 20-21, 2014*, pages 87–100, 2014. doi:10.1145/2535838.2535876.
- 7 Michel Bodin. *Certified semantics and analysis of JavaScript*. PhD thesis, Université de Rennes, 2017.
- 8 Facebook, Inc. React: A JavaScript library for building user interfaces. <https://www.reactjs.org/>, 2019. Accessed: 2019-12-19.
- 9 Philippa Gardner, Sergio Maffeis, and Gareth David Smith. Towards a program logic for JavaScript. In *Proceedings of the 39th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2012, Philadelphia, Pennsylvania, USA, January 22-28, 2012*, pages 31–44, 2012. doi:10.1145/2103656.2103663.
- 10 Arjun Guha, Claudiu Saftoiu, and Shriram Krishnamurthi. The essence of JavaScript. In Theo D’Hondt, editor, *ECOOP 2010 - Object-Oriented Programming, 24th European Conference, Maribor, Slovenia, June 21-25, 2010. Proceedings*, volume 6183 of *Lecture Notes in Computer Science*, pages 126–150. Springer, 2010. doi:10.1007/978-3-642-14107-2_7.
- 11 David Herman and Cormac Flanagan. Status report: specifying JavaScript with ML. In *Proceedings of the 2007 workshop on Workshop on ML*, pages 47–52. ACM, 2007.
- 12 Facebook Inc. React native: Learn once, write anywhere. <https://facebook.github.io/react-native/>, 2019. Accessed: 2019-12-19.
- 13 Matthew C. Loring, Mark Marron, and Daan Leijen. Semantics of asynchronous JavaScript. In *Proceedings of the 13th ACM SIGPLAN International Symposium on on Dynamic Languages, Vancouver, BC, Canada, October 23 - 27, 2017*, pages 51–62, 2017. doi:10.1145/3133841.3133846.
- 14 Magnus Madsen, Ondrej Lhoták, and Frank Tip. A model for reasoning about JavaScript promises. *PACMPL*, 1(OOPSLA):86:1–86:24, 2017. doi:10.1145/3133910.

- 15 Magnus Madsen, Ondřej Lhoták, and Frank Tip. A semantics for the essence of react. Technical Report CS-2020-03, University of Waterloo, 2020. URL: <https://cs.uwaterloo.ca/sites/ca.computer-science/files/uploads/files/cs-2020-03.pdf>.
- 16 Magnus Madsen, Frank Tip, and Ondrej Lhoták. Static analysis of event-driven Node.js JavaScript applications. 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 505–519, 2015. doi:10.1145/2814270.2814272.
- 17 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.
- 18 Frolin S. Ocariza Jr., Kartik Bajaj, Karthik Pattabiraman, and Ali Mesbah. A study of causes and consequences of client-side JavaScript bugs. *IEEE Trans. Software Eng.*, 43(2):128–144, 2017. doi:10.1109/TSE.2016.2586066.
- 19 Frolin S. Ocariza Jr., Karthik Pattabiraman, and Ali Mesbah. Detecting inconsistencies in JavaScript MVC applications. In *37th IEEE/ACM International Conference on Software Engineering, ICSE 2015, Florence, Italy, May 16-24, 2015, Volume 1*, pages 325–335, 2015. doi:10.1109/ICSE.2015.52.
- 20 Frolin S. Ocariza Jr., Karthik Pattabiraman, and Ali Mesbah. Detecting unknown inconsistencies in web applications. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering, ASE 2017, Urbana, IL, USA, October 30 - November 03, 2017*, pages 566–577, 2017. doi:10.1109/ASE.2017.8115667.
- 21 Daejun Park, Andrei Stefanescu, and Grigore Rosu. KJS: a complete formal semantics of JavaScript. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15-17, 2015*, pages 346–356, 2015. doi:10.1145/2737924.2737991.
- 22 CACM Staff. React: Facebook’s functional turn on writing JavaScript. *Commun. ACM*, 59(12):56–62, 2016. doi:10.1145/2980991.

Test-Case Reduction via Test-Case Generation: Insights from the Hypothesis Reducer

David R. MacIver 

Imperial College London, United Kingdom
david@drmaciver.com

Alastair F. Donaldson 

Imperial College London, United Kingdom
alastair.donaldson@imperial.ac.uk

Abstract

We describe *internal test-case reduction*, the method of test-case reduction employed by Hypothesis, a widely-used property-based testing library for Python. The key idea of internal test-case reduction is that instead of applying test-case reduction *externally* to generated test cases, we apply it *internally*, to the sequence of random choices made during generation, so that a test case is reduced by continually re-generating smaller and simpler test cases that continue to trigger some property of interest (e.g. a bug in the system under test). This allows for fully generic test-case reduction without any user intervention and without the need to write a specific test-case reducer for a particular application domain. It also significantly mitigates the impact of the *test-case validity* problem, by ensuring that any reduced test case is one that could in principle have been generated. We describe the rationale behind this approach, explain its implementation in Hypothesis, and present an extensive evaluation comparing its effectiveness with that of several other test-case reducers, including C-Reduce and delta debugging, on applications including Python auto-formatting, C compilers, and the SymPy symbolic math library. Our hope is that these insights into the reduction mechanism employed by Hypothesis will be useful to researchers interested in randomized testing and test-case reduction, as the crux of the approach is fully generic and should be applicable to any random generator of test cases.

2012 ACM Subject Classification Software and its engineering → Software testing and debugging

Keywords and phrases Software testing, test-case reduction

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2020.13

Category Tool Insights Paper

1 Introduction

When generating test cases to discover bugs in a system under test (SUT), it is common to use *test-case reduction* [12, 23], where large and difficult to read test cases are transformed into smaller and more readable versions, as an aid to debugging the problems discovered. Tools for automating this process are called *test-case reducers*, or reducers for short. Test-case reducers are especially important when using random test-case generation (henceforth “random generation”), which often produces large and messy initial test cases [2, 23].

This presents a particular problem for *property-based testing* libraries [2, 1] which augment unit tests with randomly generated test cases, as each type of generated test case typically requires its own test-case reducer. When generating domain-specific types with no predefined test-case reducer, users who want test-case reduction must either write their own or use one of various approaches to generic test-case reduction which attempt to derive a suitable reducer automatically.

We present an alternative approach that we call *internal test-case reduction* (henceforth “internal reduction”), which allows one to build reduction into the generation process itself. Our presentation is based on the implementation of internal reduction in Hypothesis [18], a



© David R. MacIver and Alastair F. Donaldson;

licensed under Creative Commons License CC-BY

34th European Conference on Object-Oriented Programming (ECOOP 2020).

Editors: Robert Hirschfeld and Tobias Pape; Article No. 13; pp. 13:1–13:27

Leibniz International Proceedings in Informatics



LIPICs Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

widely¹ used Python library for property-based testing. Internal reduction has been the only supported method of test-case reduction in Hypothesis since early 2016, so we consider it to be a mature and well-established technology, but it has not previously been described in the literature and does not appear to be widely known. The aim of this paper is to explain the idea of internal reduction in detail to the research community, provide insights into how it is used within Hypothesis, and illustrate the practical pros and cons of the approach via an experimental comparison with various other test-case reducers.

The key idea of internal reduction is to manipulate the underlying source of randomness consumed by a random generator, in order to cause the generator to produce smaller test cases automatically. The final reduced test case is constructed as if the generator had been implausibly lucky and produced a small and readable test case by chance.

The advantages of internal reduction over other approaches are twofold. First, given an existing internal reducer, every test-case generator comes with test-case reduction *for free*, without the need to write an external reducer. Second, because internal reduction works by re-generating test cases, any reduced test case is one that *could* have been generated. If the generator has been carefully engineered to guarantee that all generated tests satisfy certain properties, these properties will be satisfied *automatically* by the reduced test case. This helps users avoid the *test-case validity problem* [23], where reduced test-cases fail to satisfy necessary preconditions for the test. As a result, users hardly need to know that test-case reduction exists, and can just take the fact that test cases are presented in a reduced form as a given. Users must of course still ensure that their generators only produce valid test cases, but this is a problem they must solve anyway, and in practice it is often easier to construct a valid test case than it is to verify whether an arbitrary test case is valid.

Note that reduction quality or performance are *not* included among the major advantages of internal reduction. Anecdotally, and as we will provide evidence for in Section 4.3, test-case reduction in Hypothesis produces moderately better results than that found in other property-based testing libraries, but can be a fair bit slower. Based on extensive conversations with users of Hypothesis and other property-based testing libraries, we consider the user experience benefits to be worth the performance cost (and the slightly better results to be a nice bonus on top of that), but we cannot present any particularly compelling argument for this trade off beyond that experience.

Our main contributions are:

- The key idea of internal reduction, which we cast as a *shortlex optimization* problem over the choices made during generation (Section 2).
- A description of its implementation in Hypothesis (Section 3).
- A large evaluation demonstrating that Hypothesis’s internal reduction is reasonably competitive with other test-case reducers, based on bugs found in the clang and gcc compilers by Csmith [25] (a generator of C programs), differential testing of two Python autoformatters, a set of experiments testing SymPy² (a symbolic algebra library) using TSTL (a domain-specific language for testing [10]), and a reimplementaion of a set of synthetic benchmarks for QuickCheck’s reduction that were proposed in [22] (Section 4).

We also discuss threats to the validity of our results (Section 5), related work (Section 6) and present our conclusions and future goals (Section 7).

¹ Usage is difficult to measure precisely, but it is used in thousands of open source projects and has over 100,000 downloads per week. See [18] for more details of usage.

² <https://www.sympy.org>

2 Foundations of Internal Reduction

In this section we present the key idea of internal reduction. We start with a brief account of test-case reduction more broadly (Section 2.1), then discuss how these ideas can be applied to the decisions made during generation to implicitly reduce a generated test case (Section 2.2), then finish with a worked example showing how a generated test case is transformed in the course of reduction (Section 2.3).

2.1 Test-Case Reduction Fundamentals

The starting point of test-case reduction is that we have some user-specified *interestingness test* that takes a test case and determines whether it is in some sense “interesting” – generally whether it triggers a specific bug in some system under test – and some known interesting test case. The goal is to find an interesting test case which is “more readable” than the initial one, which is typically quite large and complicated.

Exactly what counts as more readable is fairly under-defined. The ultimate goal is to improve the user’s debugging experience, but this is hard to quantify. Past work on test-case reduction has identified three key features that seem generally helpful: 1) Smaller test cases are better [12, 23], 2) Users should be able to predict what features of a test case the reducer can remove, as this allows them to infer that any remaining features in the reduced test case are important [1], and 3) Test-case reducers should ideally *normalize* their input to a canonical interesting test case for each interestingness test [9].³

In support of these goals, we find it useful to think of any given test-case reducer as having a *reduction order*: a total order over all test cases ordering them from best to worst. The goal of reduction is then to find the reduction-order-minimal interesting test case.

A normalizing reducer would always find this minimal test case, but this requires brute force enumeration, which is typically infeasible. Reducers used in practice are instead only local minimizers, making small transformations to an interesting test case and checking if the transformed version is still interesting. Typically these transformations are organized into “reduction passes” and the reducer runs until it finds an interesting test case that no pass is able to reduce further.

2.2 Internal Reduction as Shortlex Optimization

Internal test-case reduction works by manipulating the underlying “random” behaviour in random generation, so we first discuss the structure of a random generator.

A random generator can be thought of as a true random variable taking values in some domain, but in practical implementations they are otherwise-deterministic functions that take a pseudo-random number generator (PRNG) and return some value. A PRNG provides an interface that the generator requests bits from, with each bit corresponding to a nondeterministic binary decision (a “coin flip”). As the PRNG is the only source of nondeterminism, any generated test case can be deterministically recreated from these binary decisions that led to it.⁴ We call these sequences of binary decisions *choice sequences*, and view random generators as parsers of choice sequences, with the PRNG as a stream interface for reading the next bits from some underlying choice sequence.

³ In practice no test-case reducers satisfy this condition in general, and the goal is only to approximate it in common cases.

⁴ This is essentially a variant on the widely known observation that you can recreate the generated value from the seed that produced it.

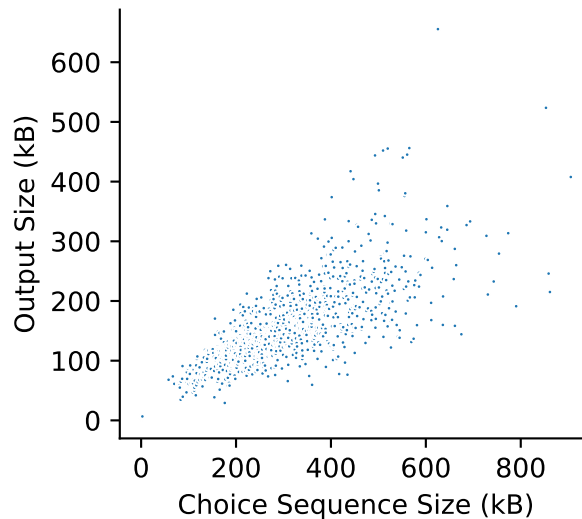
13:4 Test-Case Reduction via Test-Case Generation

A PRNG can produce infinitely many choices, but we can turn a generator into a parser of *finite* choice sequences by raising an exception when the generator reads too many bits, treating a too-short choice sequence as a parse error in the language defined by considering the generator as a parser. A generator must terminate after having made only finitely many choices, so any generated test case is the result of parsing some finite choice sequence.

Internal reduction works by performing test-case reduction not on the generated test case, but on the choice sequence that lead to it, with the hope that the test case corresponding to the reduced choice sequence is an improvement on the original. Although we cannot expect this to be true in every case, in this section we argue why, given a suitable reduction order, it is plausible that it would work for most “natural” random generators.

In our implementation of internal reduction in Hypothesis, the reduction order is the well-known shortlex order [24]: For choice sequences s and t , s is shortlex-smaller than t if $|s| < |t|$ or if $|s| = |t|$ and s is lexicographically smaller than t . Thus, internal reduction is *shortlex optimization* over the choice sequences leading to interesting generated test cases.

We now outline why this choice of shortlex order is a natural one.



■ **Figure 1** Input size vs output size for Csmith.

First, we justify that reducing the length of a choice sequence will typically reduce the size of the corresponding generated test case. This is fairly intuitive: Any part of the generated test case has to be constructed by the generator, and this will usually involve a series of nondeterministic choices, so parts of the test case that contribute to its size will correspond to regions of the choice sequence where they were generated. In the other direction, regions of the choice sequence correspond to decisions made during generation, so will usually appear as some part of the generated test case.

To see an example of this in practice, in Figure 1 we show the relationship between choice sequence length and generated program size in bytes for Csmith [25], a widely used generator of C programs. Here, the Pearson’s correlation coefficient between choice sequence and test case size is 0.73, i.e. the length of the choice sequence is a strong but not perfect predictor of the test case size.

The relationship between choice sequence and test case size can break down for a number of reasons. For example, it is common to use *rejection sampling* during generation. In rejection sampling, one retries generating some value until it satisfies some predicate. For

example to generate a number between 0 and 9 one might generate a 4 bit integer (which will be between 0 and 15) and discard the generated integer and try again if it is greater than 9. This rejection process may in principle be repeated many times, which can result in many choice sequences of different sizes, all producing the same value.

An additional common example where choice sequence size is not reflected by output size is that integers will typically be generated as a fixed number of bits. We might, reasonably enough, want reduction to reduce integers towards zero (and doing so will reduce the size of their text representation), but all possible values have the same size of underlying choice sequence, so this reduction cannot be made by reducing the number of bits drawn, only by changing their contents.

This example informs what our reduction order should be between two choice sequences of the same length. If we want fixed width integers to reduce towards zero then, depending on whether we draw these integers in big or little endian order, choice sequences that differ only in regions corresponding to a single integer should be ordered based on either the lexicographic or co-lexicographic (i.e. lexicographic from right to left) order for that region.

It is natural to extend this to the whole choice sequence, suggesting that among choice sequences of the same length we should prefer either the lexicographically or co-lexicographically smaller of the two. The choice between the two is fairly arbitrary, but we picked the lexicographic ordering in Hypothesis because it corresponds with the “time ordering” of random generation, by prioritizing decisions made earlier in the generation process, as they potentially have more impact on the generated test case.

2.3 Shortlex Optimization by Example

We now show a worked example of how the generated test case might change as the underlying choice sequence is reduced through a series of local shortlex optimization. The transformations we will show in this section do come from an actual run of Hypothesis, but we defer discussion of how these specific transformations might have been chosen to Section 3.

Our example is as follows: Suppose we have a system under test (SUT) that takes binary trees as inputs, and that it crashes when given a height imbalanced tree (i.e. some branch of the tree has two children whose heights differ by more than one).

We could test this SUT using the Python code of Figure 2 to randomly generate inputs to it. After running the SUT against several generated inputs, we might discover the tree shown along with its associated choice sequence at the top-left of Figure 3.

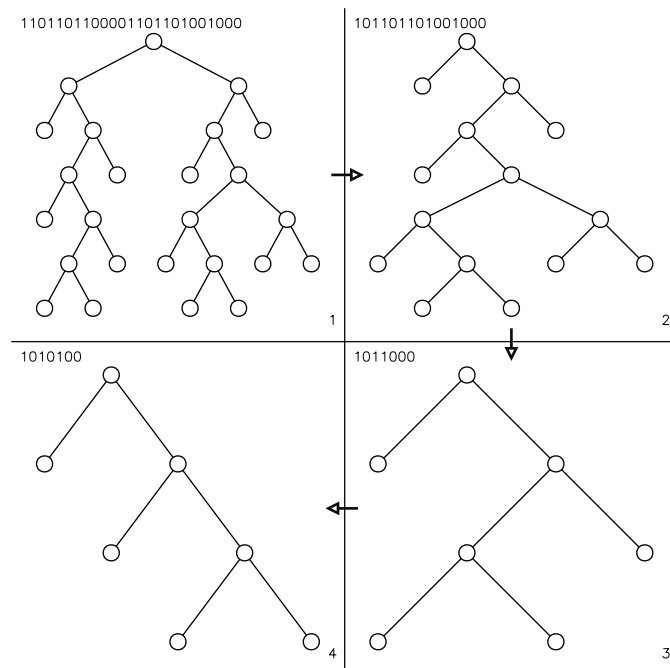
```

1 class Tree(Generator):
2     def do_draw(self, source):
3         if source.getbits(1):
4             return Branch(source.draw(self), source.draw(self))
5         else:
6             return Leaf()

```

■ **Figure 2** A simple binary tree generator. This code assumes `Branch` and `Leaf` classes for internal and leaf nodes. For ease of presentation, this generator has expected infinite size; a better one would be slightly leaf-biased.

This initial tree is moderately complicated, so we wish to find a smaller, simpler, tree, that will help us understand this bug. Rather than using *external* reduction, operating on the trees themselves, our *internal* reducer instead transforms the choice sequences producing them. We show these choice sequences in Figure 3, along with the corresponding trees produced when running the generator of Figure 2 on them.



■ **Figure 3** Successive reductions of choice sequences leading to unbalanced trees.

These transformations proceed as follows: Starting from our initial randomly generated choice sequence, labeled 1, the reducer performs the transformations $1 \rightarrow 2 \rightarrow 3$ by replacing long sequences of bits with shorter sequences of zero bits, first transforming “1101101100001...” to “101...”, then “101101101001000” into “10110000”. These transformations correspond to collapsing a subtree into a single leaf node, but we emphasize that the transformations operate on the underlying choice sequences, without reference to generated data. Finally, in the transformation $3 \rightarrow 4$, the reducer swaps two bits, transforming “10110000” into “10101000”. This swaps two subtrees, but once again is performed without any knowledge of the SUT’s data domain.

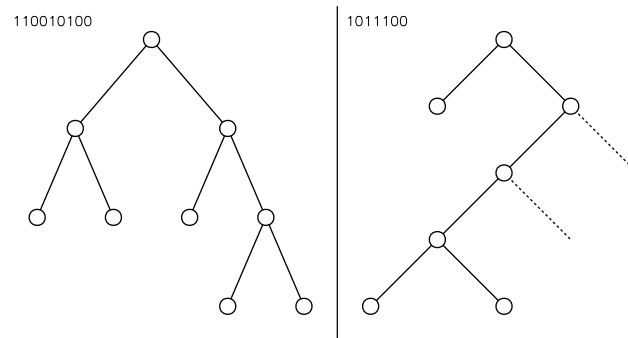
In this case, the reducer in fact finds the shortlex minimal choice sequence leading to an unbalanced tree, which can be seen in quadrant 4 of Figure 3, although in general the result will only be locally minimal.

In the course of finding these transformations, the reducer will have tried many other “failed reductions” – choice sequences that did not yield interesting test cases, either due to generating balanced trees or providing too few bits for generation of a complete test case to succeed (the left and right examples of Figure 4 respectively).

3 The Design of the Hypothesis Reducer

In this section we outline some interesting details of the Hypothesis reducer, and the rationale behind them. It will likely be of greatest interest to readers who want to learn about the intricate details of implementing a test-case reducer.⁵ Readers who care more about our high level claims may wish, at least initially, to simply regard the Hypothesis reducer as a black box and to skip to Section 4 for our evaluation of its effectiveness.

⁵ The *very* interested reader may also wish to consult the source code, which is self-contained and reasonably well documented. <https://github.com/HypothesisWorks/hypothesis/blob/master/hypothesis-python/src/hypothesis/internal/conjecture/shrinker.py>



■ **Figure 4** Some failed choice sequences arising during the reductions in Figure 3. The dashed lines represent branches that could not be generated due too short choice sequences.

3.1 A Summary of Reduction Passes

The Hypothesis reducer follows the common pattern of dividing reduction into different passes, each of which perform different classes of transformation designed to reduce interesting test cases in the shortlex order. We now provide a brief summary of these passes.

In Hypothesis 5.15.1 (which was recent at the time of this writing), the reducer contains 15 passes consisting of:

1. Six passes that delete contiguous regions of the choice sequence.
2. A pass that replaces a contiguous region of the choice sequence with a sub-region.
3. A pass for replacing a contiguous region of the choice sequence with a, possibly shorter, zeroed sequence of choices.
4. Four passes for pure lexicographic reduction.
5. Three passes for common patterns that require simultaneously lexicographically reducing some parts of the choice sequence while deleting others.

These passes tend to accumulate organically over time, based on examples we encounter that we feel the reducer should be able to handle and can't. Several of them are quite specific, but most are generic, and the combination seems to produce good results on most generators we encounter. The most specific of these by far is that one of the lexicographic passes is entirely a special case for Hypothesis's floating point generator. We discuss this further in Section 3.3.

3.2 Generator-directed Reduction

One of the biggest obstacles with test-case reduction on sequences (e.g. choice sequences for internal reduction, or file-based external test case reducers) is finding transformations that preserve some sense of syntactic validity, as syntactically invalid test cases will rarely be interesting. A classic technique here is hierarchical delta debugging (HDD) [19], which uses a grammar to find regions to delete.

In comparison to formats designed for human consumption, the choice sequence format is relatively forgiving – the only way a choice sequence can be invalid is for it to be too short.⁶ This gives the reducer a reasonable amount of leeway in making changes, and often allows it to find valid reductions by accident when some change at the choice sequence level makes essentially arbitrary changes to the generated test case.

⁶ In Hypothesis a generator may also explicitly declare a choice sequence to be invalid. We have omitted details of this for clarity of presentation.

```

1 class Generator(object):
2     def do_draw(self, context):
3         raise NotImplementedError()
4
5
6 class Source(object):
7     def __init__(self, prefix=()):
8         """A Source object such that the i'th
9         call to getbits returns prefix[i] (possibly
10        truncated) and after that is random."""
11        self.prefix = prefix
12
13        # Records the bits drawn
14        self.record = []
15        self.draw_stack = []
16        # Records (start, end) positions for draws
17        self.draws = []
18
19    def getbits(self, n):
20        """Returns an n-bit integer."""
21        i = len(self.record)
22        if i < len(self.prefix):
23            result = self.prefix[i] & ((1 << n) - 1)
24        else:
25            result = random.getrandbits(n)
26        self.record.append(result)
27        return result
28
29    def draw(self, gen):
30        """Returns the result of gen.do_draw(self)"""
31        self.draw_stack.append(len(self.record))
32        result = gen.do_draw(self)
33        self.draws.append((
34            self.draw_stack.pop(), len(self.record)))
35        return result

```

■ **Figure 5** A simplified implementation of the Hypothesis API.

Set against this, many transformations that make perfect sense at the level of the generated test case may be highly non-obvious at the level of the choice sequence without additional information about how it will be used. As we saw in the binary tree example of Section 2.3, we might be able to collapse a subtree into a single leaf by replacing some sequence of bits with a single zero bit. However, there are $O(n^2)$ possible contiguous subsequences of the choice sequence, and if we don't know how long a sequence of 0 bits to use this adds an additional $O(n)$ possibilities, giving $O(n^3)$ transformations to consider for what we would be $O(n)$ in the size of the tree for an external reducer!

If the reducer had structural information about what regions of the choice sequence corresponded to a subtree, it could similarly restrict its attention to only $O(n)$ suitable regions of the choice sequence. The key observation that Hypothesis uses to get access to this boundary information is that although we do not have a *grammar* for the language, we do have a *parser* – the generator itself – and by instrumenting the API it uses we can implement something akin to HDD, allowing us to discover transformations of the choice sequence that would be difficult to discover otherwise.

We outline this instrumented API in Figure 5. Generators are constructed as an instance of a `Generator` class, which are passed to a `draw` method on a `Source` object. The `Generator` object records the results of `getbits` calls and how these correspond to `draw` calls, which

```

1 def zero_draw(source):
2     """Attempt to replace regions corresponding to draw calls with
3     sequence of all zero bits, if doing so would not increase the length."""
4
5     i = 0
6     while i < len(source.draws):
7         u, v = source.draws[i]
8         prefix = source.record[:u]
9         suffix = source.record[v:]
10
11        # Attempt to replace the draw with a zero sequence of the same length
12        attempt = Source(prefix + [0] * (v - u) + suffix)
13        if is_interesting(attempt) and len(attempt.record) <= len(source.record):
14            source = attempt
15        else:
16            # If the number of bits was wrong, try again with the right number.
17            u2, v2 = attempt.draws[i]
18            if v2 < v:
19                attempt = test_function(prefix + [0] * (v2 - u2) + suffix)
20                if (
21                    is_interesting(attempt) and
22                    len(attempt.record) <= len(source.record)
23                ):
24                    source = attempt
25            i += 1
26    return source

```

■ **Figure 6** Replacing a draw with all zero bits.

can be used to suggest modifications to the choice sequence. In particular, for our recursive generator of Figure 2, each subtree corresponds to a single `draw` call whose start and end points are recorded on the `Source`.

A useful analogy is to consider the `draw` calls as defining the grammatical structure of the choice sequence format, while the `getbits` calls define the lexical structure. This structure often allows us to make transformations at the choice sequence level that naturally mirror the ones that a dedicated external reducer would have made to the generated test cases, without knowing any further details about what those generated test cases are.

In Figure 6 we present Python pseudo-code that shows how a reduction pass might try to replace all `draw` calls with a (possibly shorter) sequence of zero bits, one of the passes we mention in Section 3.1. Unlike the brute force $O(n^3)$ approach, running this pass attempts only $O(n)$ possible transformations⁷.

In our worked example in Section 2.3, the pass of Figure 6 is what allows us to replace any subtree with a leaf: e.g. First it might try transforming “1101101100001...” to “1000000000001...”, which would produce a valid but uninteresting choice sequence, and then it would observe that fewer choices were made in the target draw than expected, so it would try again with the single zero bit that was used, leading to the sequence “101...” that we saw in Figure 4.

⁷ This assumes that every `draw` call contains at least one `getbits` call, but where this is not the case the results can be cached, a detail we omit here.

3.3 Generator / Reducer Co-design

There is a certain amount of co-design between Hypothesis’s library of generators and its reducer. We show in Section 4.1.2 that this co-design isn’t strictly necessary, in that the Hypothesis reducer produces reasonable results without it, but we have nevertheless found it useful.

The co-design occurs when we encounter an example that reduces poorly, requiring us to modify one or both of the generator or the reducer. Typically, when the example is user provided we will modify the reducer, and when it is part of the Hypothesis library of generators, we will modify the generator to be more “reduction friendly”, but in some cases it is still better handled by modifying the reducer.

In particular, as we mention in Section 3.1, there is a special case for our floating point generator. This generator is designed so that lexicographic reduction will produce “visually simpler” floating point numbers. This is important because if a float was generated as its IEEE representation it would instead reduce towards 0.0, which tends to produce reduced test cases that look pathological. e.g The most reduced non-zero double precision float would be $5e-324$, when ideally we would like to reduce non-zero floats to 1.0.

This results in certain transformations that look very natural to a human reader but are quite complicated at the choice sequence level. e.g. 9.0 is represented as a 64-bit integer value of 9 in our internal float encoding, but 9.1 is represented as 9237896145653045656. Although going from the latter to the former is an obvious reduction to a human reader, and is a lexicographic reduction at the choice sequence level, it would be quite hard for the reducer to discover on its own. As a result, it was worth adding a special case to our implementation to make it aware of transformations that were obvious at the floating point level but not at the underlying choice sequence level.

The floating point generator is the only case we’ve encountered that required this level of special casing, and this was largely only needed due to the relative complexity of the floating point format. Additionally, it was only worth it because it is such a foundational generator: If it had not been part of our core library, it would likely not have been worth investing much time in it, and so it would have been left with the default behaviour which, while suboptimal, was still relatively adequate.

More commonly, it is worth designing core generators to aid the performance of test-case reduction, because some designs make it easier to find relevant reductions. Users are not expected to need to do this, but the cost-benefit trade off is different for the core Hypothesis library of generators, as they are more widely used and we have greater expertise in the behaviour of the reducer.

To illustrate this, in Figure 7 we show an example of how one might⁸ generate lists using Hypothesis. This generator arranges matters so that an element of the list can be deleted by deleting a contiguous region of the choice sequence, corresponding to the `getbits` call followed by a subsequent `draw`. Deleting the region corresponding to these two calls effectively causes the loop to skip over the iteration where the generated element would previously have been added.

In contrast, if we generated lists by first drawing a length parameter and then drawing that many elements, deleting an element of the list would require first lowering that length parameter and then deleting a later part of the choice sequence. Identifying all such pairs would require $O(n^2)$ transformations.

⁸ For simplicity, this generator elides details which control the expected size of the list. The real version also contains a hint to the reducer about what regions are worth deleting.

```
1 class ListGenerator(Generator):
2     def __init__(self, elements):
3         self.elements = elements
4
5     def do_draw(self, data):
6         results = []
7         while True:
8             more = data.getbits(1)
9             if more:
10                results.append(data.draw(self.elements))
11            if not more:
12                break
13        return results
```

■ **Figure 7** A simplified list generator.

Hypothesis does in fact have a reduction pass that does this, because such patterns are common in user code, but its performance is comparatively poor due to the large number of transformations to be tried, and so we have used the implementation that allows for more efficient reduction.

An additional benefit of this is that, because it is relatively easy to transform the choice sequence in ways that preserve the structure of the generated list, other reductions become possible. For example, due to trying to delete short subsequences of the choice sequence, when generating lists of lists, Hypothesis will try merging adjacent lists (e.g. transforming $[[1, 2], [3, 4]]$ into $[1, 2, 3, 4]$), because this corresponds to deleting the choices in two adjacent calls to `getbits`.

4 Case Studies and Experiments

In this section we present data on internal reduction in Hypothesis, comparing the cost of reduction and final size of reduced test cases to those of existing external reducers.

Our goal is not to show that internal reduction is especially impressive on these metrics. As we discuss in Section 1, the primary benefits of internal reduction are not its performance or the quality of the end results, but that it provides adequate reduction for any generated test case, while avoiding the test-case validity problem. As such our evaluation is mostly intended to be descriptive, and to increase the plausibility of our claim of adequacy.

We structure our evaluation around the following research questions:

1. How does the size of the final test case obtained through internal reduction compare to that obtained through external reduction? (RQ1)
2. How expensive is internal reduction compared to external? (RQ2)
3. How much overhead does the process of going through the generator introduce? (RQ3)

In addressing these research questions we primarily focus on future-proof metrics that are independent of our particular experimental setup: the number of SUT and generator invocations. Unlike the metrics, wall clock time is sensitive to specific implementation choices in Hypothesis and the tools against which we compare, and other engineering issues such as the choice of implementation language. Furthermore, to make our large study feasible, experiments were performed in parallel on a multi-core machine, with associated impact on wall clock time variance.

Our main three evaluations use Hypothesis to find and reduce real bugs in three classes of real world software:

13:12 Test-Case Reduction via Test-Case Generation

- We used a modified version of Csmith to allow Hypothesis to generate C programs, which we used to trigger bugs in old versions of the open source C compilers, gcc and clang (Section 4.1);
- We wrote a custom generator of Python programs and used it to perform differential testing of yapf [7] and black [17], two open source Python autoformatters (also Section 4.1);
- We implemented a Hypothesis-based test harness to trigger bugs in SymPy, an open source symbolic algebra library (Section 4.2).

For completeness, we also compared Hypothesis on a series of synthetic benchmarks used in [22] to evaluate SmartCheck, a proposed generic test-case reducer for QuickCheck (Section 4.3).

We note that while we have been able to apply Hypothesis to this relatively diverse range of applications, in order to compare with a number of different test-case reduction tools, no one of the tools that we compare with could be easily applied to all of these case studies. This is an important selling point for internal reduction: it works at the level of choice sequences, and any randomized generator can be relatively easily adapted to consume a choice sequence instead of using a pseudo-random number generator, thus internal reduction has wide applicability.

We have made the code for reproducing the data for these experimental results available at <https://github.com/mc-imperial/hypothesis-ecoop-2020-artifact>.

4.1 Evaluation on Generated Programs

We designed a system for running controlled reduction experiments on Hypothesis-generated examples and used it to run tests on real world bugs found by two different program generators:

1. A patched version of Csmith,⁹ which uses Hypothesis as its source of entropy, where any calls to methods named `make_random` were wrapped in a macro so as to show up as if they were a generator passed to `draw`.
2. A generator of syntactically valid Python programs that we wrote ourselves using Hypothesis's library of generators.

For each of these generators we wrote interestingness tests that would use the generated test cases to look for bugs in some real world software. For Csmith-generated programs, these were crashes or wrong code bugs in old versions of gcc and clang. For Python programs produced by our generator, we used them to test a Python autoformatter, yapf, and checked its output for style violations.

For each of these generators and their corresponding interestingness test, we built a corpus of 200 choice sequences that resulted in interesting test cases. We then ran reduction for each of these starting points using each of: 1) Internal reduction provided by Hypothesis; 2) *C-Reduce* [23], a test-case reducer primarily designed for C programs but suitable for any text format; 3) *Picire* [13], a modern implementation of the classic delta-debugging algorithm.

We explain this experimental setup in more detail in Section 4.1.1, and then present the results in Section 4.1.2

⁹ <https://github.com/HypothesisWorks/csmith>

4.1.1 Experiment Design

For each experiment we defined a class of bugs we were looking for, with precise interestingness tests for identifying each possible bug.

For the generator of Python programs, we used it to perform differential testing of yapf, a Python source code autoformatter developed at Google, against black, a more recent and more widely used autoformatter. The test we performed was that we ran black on the generated source, followed by pycodestyle,¹⁰ a style checker for conformance to PEP8, the official Python style guide. If there were no style errors, we then ran yapf on the black-formatted source. Any style errors introduced constituted a bug in yapf, as it had taken source code that it was possible to format correctly and introduced a style violation.

For Csmith, we ran a large number of old versions of gcc and clang,¹¹ at four different optimization levels (-O0, -O1, -O2, -Os). This could produce three distinct types of bug: The compiler could crash, the compiled binary could crash when run, or there could be a miscompilation, determined when the output differed from that on gcc 8.3.0 (the latest of the compilers tested) compiled at -O0. Whenever an example triggered multiple bugs we associated it with the bug it triggered in the latest compiler, at the lowest optimization level for that compiler, as this seemed like a reasonable proxy for how interesting the bug was.

The need for a validity oracle for C-Reduce and Picire. Csmith guarantees generating C programs that are free from undefined behaviour by construction [25]. When we drive Csmith via Hypothesis, test-case reduction involves using Csmith to generate successively simpler programs, each of which is thus free from undefined behaviour by construction. In contrast, neither C-Reduce nor Picire provides such a guarantee. In order to use these reducers we had to define a *validity oracle* that detected if the program was likely to be free from undefined behavior. The validity oracle that we used compiled the program with clang and GCC and checked for warnings likely to indicate undefined behavior, as recommended in the C-Reduce documentation,¹² and in addition ran the generated binary under UBSan¹³ to look for non-trivial undefined behavior that was only detectable at run time. We did not apply the validity oracle when reducing compiler crash bugs, as the execution result of the program is not relevant in such cases.

We generated a corpus for each experiment by sampling choice sequences of length up to 8KB (Hypothesis's default maximum size) until we had 200 choice sequences that triggered bugs for each experiment. For the Python generator, this small buffer size was not a problem, but for Csmith this was a significant restriction – when generating a corpus without this size restriction we found only about 2% of choice sequences corresponding to programs triggering bugs were under 8KB. This corroborates previous observations in [25] that Csmith is most effective when generating large programs. We attempted to run the Csmith experiment with a larger buffer size, but unfortunately Hypothesis is not currently well designed for larger sizes and we hit some memory limitations, so we decided to restrict ourselves to examples within Hypothesis's normal operational parameters.

For each corpus member and each reducer, we ran the reduction to completion, instrumented so as to record SUT calls and report on successful reductions.

¹⁰<https://pycodestyle.pycqa.org/en/latest/>

¹¹All of those installed by https://github.com/mattgodbolt/compiler-explorer-image/blob/master/update_compilers/install_compilers.sh. This included gcc versions ranging from 4.1.2 to 8.3.0 and clang versions ranging from 3.9.1 to 7.0.0, but not every patch release in that range.

¹²<https://embed.cs.utah.edu/creduce/using/wrong1/test1.sh>

¹³<https://clang.llvm.org/docs/UndefinedBehaviorSanitizer.html>

	Csmith	Python
No Reduction	1963.9 (1750.3–2230.2)	417.2 (365.8–483.0)
C-Reduce	120.0 (114.2–126.2)	70.9 (64.5–76.8)
Hypothesis	812.3 (786.8–843.1)	71.8 (64.7–78.8)
Picire	345.09 (321.3–375.5)	75.7 (68.8–82.4)

■ **Figure 8** Mean sizes, measured as number of bytes after formatting, of final examples for each reducer.

4.1.2 Experimental Analysis

In order to answer RQ1, we have to define a suitable notion of size. The number of bytes is the obvious choice, but one subtlety to consider is that many size reductions are both impossible in internal reduction, and also undesirable! For example, removing whitespace is often a valid reduction in size that reduces readability. In order to offset this, instead of raw size we consider formatted size. For each experiment we used a standard automatic formatter, `clang-format`¹⁴ for C programs and `black` for Python programs, and consider the size of the formatted result. We also strip comments from the C programs.

We justify this as a reasonable metric by observing that the purpose of test-case reduction is not actually to reduce size, but rather to ease debugging. A formatter is designed to improve the readability of the code (and it is often worth formatting reduced test cases to understand them better), and a human reader is unlikely to pay attention to the comments unless they are an aid to understanding, so this is a truer representation of the size a human reader sees.

We calculated the size of the reduced test case for each test case and reducer, and report the mean size in Figure 8 alongside 95% bootstrap confidence intervals. A permutation test for difference of means shows that the differences between these means are significant for all three reducers on the Csmith experiment ($p < 10^{-5}$ for C-Reduce vs each of the others, $p \approx 0.0003$ for Hypothesis vs Picire), and non-significant at a threshold of 0.05 for all pairs on the formatting example.

We discuss RQ1 separately in the context of the Python formatting experiments and the Csmith experiments, as the findings are substantially different.

Python formatting results for RQ1. Figure 8 shows that, for the Python formatting case study, Hypothesis, Picire and C-Reduce perform comparably well (with overlapping confidence intervals regarding reduced test case size, and nonsignificant differences in means). As we discuss above, our claim is that internal reduction should work well enough to be useful, not that it should out-perform other reduction approaches, and these results support that claim.

Csmith results for RQ1. The results of Figure 8 show that C-Reduce and Picire are able to achieve substantially smaller reduced programs than Hypothesis on average. Regarding our aim that internal reduction should be good enough to be useful: the reduction factors associated with Hypothesis in Figure 8 would certainly be worth having for debugging purposes if no other reducer were readily available, and the fact that test cases retain the Csmith guarantee of validity when reduced using Hypothesis is a potentially important bonus (especially for wrong code bugs) that the size results of Figure 8 do not show.

¹⁴<https://clang.llvm.org/docs/ClangFormat.html>


```

1 #include "csmith.h"
2 static long __undefined;
3 static int8_t func_1(void);
4 static int8_t func_1(void) {
5     int8_t l_2 = 0L;
6     return l_2;
7 }
8 int main(int argc, char *argv[]) {
9     int print_hash_value = 0;
10    if (argc == 2 && strcmp(argv[1], "1") == 0)
11        print_hash_value = 1;
12    platform_main_begin();
13    crc32_gentab();
14    func_1();
15    platform_main_end(crc32_context ^ 0xFFFFFFFFFUL, print_hash_value);
16    return 0;
17 }

```

■ **Figure 9** The minimal size Csmith program that Hypothesis could find, which is essentially the smallest program that Csmith can generate.

Nevertheless, Hypothesis does produce substantially larger reduced test cases than the external reducers, and the reasons for this provide various insights into the limitations of internal reduction.

The first reason to note is that Csmith-generated C programs have a certain amount of “necessary size”, due to boiler plate that every Csmith program contains. Because internal reduction reduces test cases by re-generating them, the minimum size of the reduced test cannot be lower than the smallest test the generator can produce.

Effectively, Hypothesis is reducing against a harder validity oracle: It has to produce an interesting test case that Csmith could have generated, while C-Reduce and Picire merely have to produce an interesting test case that is a valid C program which appears free of undefined behaviour. For most uses of Hypothesis in property-based testing, this sort of constraint is mild and perhaps actively desirable, but in this case it results in a significantly larger final test case. In particular, it is impossible for Hypothesis to prevent Csmith from generating its standard boiler plate code, so there is a certain baseline difference between Hypothesis and an external reducer that it can never do better than.

In order to determine this baseline, we ran Hypothesis on each starting example, reducing the choice sequence subject only to the constraint that it successfully generates a program. The smallest program found by Hypothesis during these reductions is shown in Figure 9, which we know (from familiarity with how Csmith works) is essentially the smallest program Csmith is capable of generating. This gives us a baseline minimum size for Hypothesis reduced programs of 410 bytes. In contrast, C-Reduce and Picire are perfectly capable of producing an empty file, or a trivial 14-byte main function definition if we require that the file can produce an executable (which we do for wrong code bugs). This already accounts for a sizable proportion of the difference in size. Adjusting for these baselines, Hypothesis generates a mean final size of around 402 bytes, and Picire of around 331 bytes. This difference is still statistically significant, but much more reasonable.

In order to understand the size difference above and beyond this baseline, we ran C-Reduce on the Hypothesis final example for the smallest examples of a crashing bug and a wrong code bug respectively. We show examples of these in Figure 10 and 11.

13:16 Test-Case Reduction via Test-Case Generation

The difference on the crash bug, where C-Reduce is less constrained, largely comes from the larger baseline we discuss above, while the difference on the wrong code bug demonstrates a number of other issues: Csmith will always pre-declare union definitions, and uses long identifiers, both of which C-Reduce is able to fix.

We also see an advantage of Hypothesis in this example: In Figure 11, C-Reduce has produced a call to `printf` with too many arguments. This is defined behavior, as the extra arguments are ignored, but is suspicious and likely to be distracting when debugging. In contrast, the reduced program produced by Hypothesis is Csmith-generated, and has no such issues.

Hypothesis reduced:

```
1 #include "csmith.h"
2 static long __undefined;
3 static const volatile int32_t g_2 = (-1L);
4 static const int8_t func_1(void);
5 static const int8_t func_1(void) {
6     volatile int8_t l_3 = (-1L);
7     l_3 = g_2;
8     return g_2;
9 }
10 int main(int argc, char *argv[]) {
11     int print_hash_value = 0;
12     if (argc == 2 && strcmp(argv[1], "1") == 0)
13         print_hash_value = 1;
14     platform_main_begin();
15     crc32_gentab();
16     func_1();
17     transparent_crc(g_2, "g_2", print_hash_value);
18     platform_main_end(crc32_context ^ 0xFFFFFFFFFUL, print_hash_value);
19     return 0;
20 }
```

C-Reduce run on the Hypothesis output:

```
1 #include "csmith.h"
2 const volatile a;
3 b() { volatile int8_t c = a; }
```

■ **Figure 10** The smallest Hypothesis reduced crash bug.

To emphasise the above discussion regarding Csmith boiler plate: in both of these examples we can see that Hypothesis is quite close to being constrained by a fundamental limitation of this approach. The limitation is not its ability to reduce further (although in Figure 11 we do see what is likely a missed reduction at the choice sequence level – the third field of the union is successfully removed by C-Reduce but not by Hypothesis even though it plausibly could be), but the fact that it guarantees that reduced examples are ones that could be generated means that the reduced examples must include certain features that Csmith will always generate: e.g. variables will always be initialized, functions will always be pre-declared, and union types are always declared separately from their usage.

These larger sizes are certainly a minor downside of internal reduction, in that for ease of debugging a smaller program is usually more useful. That said, in the case of Csmith, being limited to reducing to programs that Csmith can generate means that even reduced tests will be executable programs that follow a well-known structure and are guaranteed to be free from undefined behaviour, which might make them ideal as end-to-end tests for addition to a compiler regression test suite (rather than e.g. tests that simply check whether a compiler crashes, but that are otherwise meaningless). It is also arguable that since most of the extra size is easy-to-understand boiler plate, its presence has little practical significance.

Hypothesis reduced:

```

1  #include "csmith.h"
2  static long __undefined;
3  union U1 {
4      const int32_t f0;
5      const unsigned f1 : 17;
6      const volatile signed : 0;
7  };
8  static const union U1 g_2 = {-1L};
9  static const union U1 func_1(void);
10 static const union U1 func_1(void) { return g_2; }
11 int main(int argc, char *argv[]) {
12     int print_hash_value = 0;
13     if (argc == 2 && strcmp(argv[1], "1") == 0)
14         print_hash_value = 1;
15     platform_main_begin();
16     crc32_gentab();
17     func_1();
18     transparent_crc(g_2.f0, "g_2.f0", print_hash_value);
19     transparent_crc(g_2.f1, "g_2.f1", print_hash_value);
20     platform_main_end(crc32_context ^ 0xFFFFFFFFFUL, print_hash_value);
21     return 0;
22 }

```

C-Reduce run on the Hypothesis output:

```

1  #include "csmith.h"
2  union {
3      int32_t a;
4      unsigned b : 17;
5  } c = {-1L};
6  int main() {
7      printf("%d\n", c.a, c.b);
8      return 0;
9  }

```

■ **Figure 11** The smallest Hypothesis reduced wrong code bug, with additional reduction provided by C-Reduce.

Either way, RQ1 has a clear answer on the Csmith experiments: Hypothesis produces examples in the same order of magnitude as, but still substantially larger than, those produced by specialized reducers such as C-Reduce that are well-adapted to the problem domain, but produces of examples of comparable but slightly larger size to those found by more generic reducers.

To evaluate RQ2, we recorded the number of SUT evaluations made during the run of these experiments. We show the results of this in Figure 12

Here all differences are statistically significant at $p < 10^{-5}$. Hypothesis is thus about four times faster¹⁵ than Picire on the Csmith experiment, and about 50% slower on the formatting example. We haven't investigated this in detail but expect that the latter is because Hypothesis makes a number of lexicographic transformations to the choice sequence that don't impact the final size of the generated test case, but result in e.g ensuring generated string literals contain only zeroes.

¹⁵In terms of SUT calls that is. In terms of wall clock time it was actually slower due to the high cost of how we invoked Csmith.

	Csmith	Python
C-Reduce	3968.0 (3731.8–3216.7)	863.3 (797.5–937.0)
Hypothesis	762.0 (701.8–829.6)	1284.565 (1106.56 1556.175)
Picire	3138.9 (2970.9–3348.9)	529.23 (483.61, 579.205)

■ **Figure 12** Number of SUT invocations for each reducer.

To answer RQ3, we also recorded the number of generator evaluations, and for each experiment calculated the ratio of generator evaluations to SUT calls (every generator evaluation leads to an SUT call, so the former is always larger than the latter). We calculated a 95% bootstrap confidence interval for the geometric mean of these ratios (the geometric mean being chosen as the appropriate mean to use for comparing ratios). For the Csmith experiment this gave us a confidence interval of 2.78–2.94, and for the formatting experiment the interval was 1.21–1.30. i.e. for Csmith we performed nearly three times as many generator invocations as SUT invocations, while for Python we performed up to about 30% more. The difference is likely accounted for by the fact that the Python generator was built on top of Hypothesis’s core library of generators which as we describe in Section 3.3, are designed to behave well with lexicographic reduction in general and Hypothesis’s reducer in particular.

How much overhead this corresponds to in practice depends significantly on the generators and SUTs in question. Our interface to Csmith was quite slow, so there generation time probably dominated even without any overhead, but for most cases we would expect the generator to be significantly faster than the SUT.

4.2 Case Study: SymPy

TSTL [11] is a domain specific language defined for testing APIs written in Python. Actions using the API are described using the TSTL language, and it builds tests as sequences of actions, expressed as fragments of Python code that are evaluated against a model of the SUT.

Reduction in TSTL consists of attempting to find shorter sequences of actions that can trigger the same bug. Previously work on TSTL’s reducer [11] tested SymPy, a symbolic algebra library for Python, and we adapted these tests to use Hypothesis in order to evaluate internal reduction for this use case.

One downside of comparing with TSTL is that a TSTL test is always valid – any action which should not be run is simply ignored – so the benefit of guaranteed validity associated with internal reduction is not relevant, but it is still a reasonable point of comparison for reducer effectiveness.

4.2.1 Experiment Design

We implemented a backend that takes a TSTL-generated harness and runs it with Hypothesis, which we used to run the TSTL tests for SymPy from its examples directory. We ran these tests against version 1.1.1 of SymPy, which is slightly older than the latest version, as we knew that the test harness was capable of finding many bugs in this older version, providing us with a variety of example bugs on which to evaluate reduction.

This backend does not implement TSTL’s checks, which run a number of equivalence checks on the generated SymPy programs to assert that various expressions that are expected to give the same result do in fact do so. These checks were minimally useful for SymPy [8], and were prohibitively slow, thus they would have limited the amount of data we could have collected.

As a point of comparison, we used a custom implementation of delta debugging which we adapted to take advantage of two structural features of TSTL: It would automatically discard any actions that were no longer able to run, and prune all steps after the failing one. We did not compare to the TSTL reducer due to wanting to ensure we matched the slightly different semantics of our backend implementation, and for convenience when instrumenting it, but believe this modified delta debugging should work similarly well to its standard reducer. We did not however implement anything equivalent to its test-case normalization features [9].

To enable us to gather a large corpus of data, we aggressively pruned slow tests by removing test cases where an individual step took more than two seconds to run. This implicitly removed a large class of errors, as it appears to be very easy to trigger `RecursionError` bugs in SymPy which, for some reason (possibly a high cost associated with each recursive call), always resulted in the triggering step exceeding this timeout. This potentially impacts the generality of our results, but there were sufficiently many other errors in SymPy that it seemed unproblematic to exclude them.

Additionally, we found a number of the SymPy test cases were *flaky* – that is, they did not reliably produce the same exception when run with different random or hash seeds. We don't entirely understand why this would be the case (we expect it is something internal to SymPy's implementation) but we didn't spend a great deal of time investigating. We know from experience that flaky test cases tend to lead to poor performance in most test-case reducers, and we wished to avoid these dominating the results, so we attempted to remove any test cases where reduction passed through a flaky test case. We removed test cases where any of the original generated test case or either of the internally or externally reduced final test cases were flaky, but flakiness checking was fairly expensive so we did not check all intermediate results.

Starting from an initial generated corpus of 3000 distinct failing test cases, removing flaky tests left us with 2930 interesting test cases. These were spread across 33 distinct errors, which we distinguished based on error type and line number, and had a mean length of 64.5 (which gave a 95% confidence interval for the population mean of 63.6 – 65.4). Notably, this is somewhat larger than the mean size of 44.7 reported in [11]. While we did not investigate the cause of this in detail, there were a number of small differences in our experimental setup which could account for it, such as the exclusion of the relatively easy to trigger `RecursionError` bugs.

For each of these initial test cases, we ran both the Hypothesis reducer and our delta debugging implementation for TSTL, subject to the interestingness test that an exception was raised with the original exception type and line number. We recorded the number of SUT calls made by each, and the final size of the reduced test cases.

4.2.2 Experimental Analysis

On average (geometric mean), Hypothesis made 20.6 (95% confidence 20.3 – 21) times as many SUT calls as delta debugging, resulting in tests that were 83% (95% confidence 82% – 84%) of the size produced by delta debugging.

This is relatively expensive for a marginal gain. However, that seems to be less a feature of internal reduction and more one of the problem domain: As part of the work on test-case normalization in [9], they implemented normalization passes which performed similar external transformations to those enabled by Hypothesis's lexicographic internal reduction, and when these normalization passes were enabled reduction took about thirty times as long and obtained test cases that were about 55% of the size of those obtained without normalization.

We think it likely that the performance of Hypothesis could be substantially improved on this experiment, but resisted the urge to optimize for this use case for now, letting the experimental results stand as they are. Brief investigation suggested that Hypothesis’s heuristics for reduction pass ordering do not work very well on these examples, which lead to it doing a significant amount of lexicographic reduction when it could still have usefully been trying to reduce the size of the choice sequence.

4.3 Evaluation against QuickCheck and SmartCheck

The only previous evaluation of test-case reduction in QuickCheck we are aware of comes from [22], which defined SmartCheck, a generic reducer for algebraic data types, and introduced a set of five synthetic benchmarks to compare it to QuickCheck. Each of these benchmarks consists of some data type to generate to test some code that has a known (deliberately inserted) bug in it. We have reimplemented these benchmarks in Python to evaluate Hypothesis on them and compare its behavior to that of QuickCheck and SmartCheck.

The five benchmarks are “bound5”, “binheap”, “calculator”, “parser”, and “reverse”. We updated these from the originals to improve QuickCheck’s behavior, mainly by replacing some ineffective custom reducers with QuickCheck’s `genericShrink`. We also changed the “binheap” benchmark to add a precondition that prohibited invalid heaps, as we noticed that much of SmartCheck’s performance on that benchmark came from very rapidly reducing to small but invalid heaps (an instance of the test-case validity problem).

Experiment	Hypothesis	QuickCheck	SmartCheck
binheap	9.02 (9.01–9.03)	9.00 (9.00–9.00)	9.42 (9.37–9.48)
bound5	2.08 (2.07–2.10)	11.30 (10.91–11.76)	6.02 (5.79–6.29)
calculator	5.00 (5.00–5.00)	5.11 (5.07–5.15)	5.00 (5.00–5.01)
parser	3.31 (3.28–3.34)	3.99 (3.98–4.01)	4.08 (4.01–4.14)
reverse	2.00 (2.00–2.00)	2.00 (2.00–2.00)	2.00 (2.00–2.00)

■ **Figure 13** Mean size of reduced examples on synthetic benchmarks. Each data type has a different notion of size associated with it, but it typically means something like number of nodes in the tree.

Experiment	Hypothesis	QuickCheck
binheap	170.31 (166.14–174.76)	88.22 (86.90–89.55)
bound5	95.13 (93.57–96.91)	1438.89 (1282.34–1811.64)
calculator	72.41 (70.57–74.32)	30.97 (29.92–32.37)
parser	126.50 (124.11–128.90)	34.23 (33.63–34.81)
reverse	50.84 (50.40–51.29)	17.68 (17.27–18.10)

■ **Figure 14** Mean number of test cases tried while reducing synthetic benchmarks.

We ran each benchmark 1000 times for each library. We present the mean sizes of the reduced examples in Figure 13, and the mean number of SUT evaluations made in Figure 14. We ran into some technical difficulties obtaining the number of SUT evaluations made by SmartCheck and, as it omits many classes of transformation that both Hypothesis and QuickCheck consider (e.g. reducing the value of generated integers) and did not do particularly well on the size evaluation besides, didn’t feel it was especially useful to invest more time on the problem.

By a permutation test, all differences in mean SUT invocations are significant at $p < 10^{-5}$. For sizes, differences were significant at $p < 10^{-4}$, with the following exceptions:

- All implementations reliably produced the minimal size example for “reverse” so there was no difference in means.
- Hypothesis and SmartCheck on the “calculator” example ($p \approx 0.5$)
- Hypothesis and SmartCheck on the “parser” ($p \approx 0.02$).
- Hypothesis and QuickCheck on the “binheap” benchmark ($p \approx 0.003$).

To account for multiple testing we set a significance threshold at $p < \frac{0.05}{30} \approx 0.0017$ (by applying the Bonferonni correction – there are three pairs of comparisons for each benchmark, for each of size and SUT count, so thirty tests), so these should all be considered nonsignificant.

The only case where Hypothesis produced worse average results than QuickCheck (significant or not) was the “binheap” benchmark, where it did very slightly worse than QuickCheck (9.02 vs 9.0). We haven’t investigated why but suspect it’s due to difference in the distribution of initial test cases (Hypothesis tends to produce larger examples) rather than the reducer. Whatever the reason, the difference, though statistically significant, is tiny.

We note that the behaviour of QuickCheck on the “bound5” example is pathologically bad, both in size and performance, in large part because it was constructed to be so. Hypothesis fares well on this example without modification, showing one of the advantages of having a more sophisticated reducer by default.

Thus on RQ1 Hypothesis fares well compared to QuickCheck, generally producing similar or better results. On RQ2, Hypothesis proves more expensive than QuickCheck by a factor of 2–3, depending on the benchmark.

5 Threats to Validity

The main empirical claims of our paper are that our model of internal reduction through shortlex optimization is viable, and in particular that it provides results that are competitive with alternative reducers that might be used in its place.

As we have been using it in the context of a widely deployed testing library for more than four years, we are quite confident of its viability, at least within our application domain, and our empirical results in Section 4 support the claim that it performs reasonably with respect to alternatives.

The main threat to validity is how well these results generalize. Although we have presented four reasonably diverse case studies, three of which were on their own larger than most previous evaluations of test-case reduction, the range of software and generators used in practice is naturally larger yet. It is plausible that there are reduction problems that we have simply never run into that present their own challenges.

A common factor in all of our experiments is that the starting points were not especially large – Hypothesis by default only considers choice sequences of at most 8KB, and we retained that restriction in our analysis. As we discuss in Section 4.1.1, this was a particularly notable restriction in the case of Csmith.

Our intuition, which is backed by a certain amount of anecdotal evidence, is that most test-case reducers experience problems at larger scales that they do not see at smaller ones, because larger test cases offer more opportunities to get stuck in local minima. Additionally, often large test cases trigger bugs in SUTs that were difficult to trigger at smaller scales – either because they are intrinsically connected to test case size (a scenario that tends to reduce very poorly in general) or because they simply happen with too low probability at small sizes. Between these two factors, we expect interesting new difficulties to arise at larger scales, requiring more work on Hypothesis’s reducer.

This also points to the other major limitation of our results: Although our claim is that internal reduction as a general model is viable, our empirical results are restricted to its implementation in Hypothesis. This suffices as an existence proof, but the Hypothesis reducer has been the subject of considerable engineering effort, and our results do not determine how much of the viability of internal reduction is only because of that engineering effort.

However, part of why the Hypothesis reducer is so sophisticated is because internal reduction rewards that: Because one reducer can serve many different types of test case, it was worth investing that effort into it, and the reducer can in principle be used in many different contexts, so even if it turns out that internal reduction is only viable with this engineering work, we don't consider that to be a major point against it.

6 Related Work

There are several categories of work related to ours, which we now describe: Test-case reduction in general (Section 6.1), test-case reduction in property-based testing (Section 6.2), use of the choice sequence model to improve generation (Section 6.3), and finally other users of internal reduction (Section 6.4).

6.1 Test-Case Reduction

Our work on internal reduction naturally builds on prior work on test-case reduction.

Test-case reduction was first described in the original papers on *delta debugging* [12, 26]. Most subsequent research has been focused on continuing delta debugging's goal of reducing the size of the test case, with other reductions such as our lexicographic passes being treated as of secondary interest.

This work on reducing size has generally focused on taking advantage of the structure of particular input formats. The major examples of this in the literature include *hierarchical delta debugging* (HDD) [19], which makes use of a grammar for the test-case format, and C-Reduce [23], which is extensively specialized to features common in C and C-like languages.

As we discuss in Section 3, the Hypothesis reducer is a similarly specialized reducer designed for the class of languages parsed by generators, and its design has been inspired by this prior work. In particular, the approach we describe in Section 3.2 of marking out regions of the choice sequence corresponding to parts of the test case very strongly resembles HDD's use of a grammar to do the same, and the pass-based approach we describe in Section 3.1 strongly resembles the architecture of C-Reduce.

One exception to the prior focus on reducing size is [9], which introduced the notion of test-case normalization as an important property of reducer. Additionally, although this was not made explicit, the normalization passes suggested in [9] can be regarded as optimizing for the lexicographic ordering, which makes their approach another example of our suggested goal of shortlex optimization. However, this was in the context of an external reducer, not an internal one.

6.2 Test-Case Reduction in Property-Based Testing

Test-case reduction has been an important feature in property-based testing since the early work on QuickCheck [2]. In property-based testing, test-case reduction is usually called *shrinking*, but for consistency we will continue to use the term test-case reduction.

In the original QuickCheck, and other property-based testing libraries closely based on it, test-case reduction follows the external reduction model, with reducers run on the generated test cases once an interesting one has been discovered, with an appropriate reducer selected based on the type of the generated data or provided by a user.

Originally these reducers were hand-written ones. However, most users do not particularly want to write their own test-case reducers, so this led to the introduction of generic test-case reducers. These are particularly popular in Haskell, where most data is represented with algebraic data types, and good generic programming libraries allow for automatically deriving reducers for most data types that are “good enough” (any test-case reduction will tend to improve the utility of property-based testing, and to be worth the effort, hand-writing a reducer has to be less work than the debugging effort it saves). Indeed, the derivation of such reducers has been used to motivate the development of some of these generic programming libraries [14]. Generic reduction was also explored in [22], but the suggested approach does not appear to have been widely adopted.

However both manual and generic approaches to test-case reduction suffer a variant of the test-case validity problem: After reduction has been performed, the final reduced test case may be one that could not have been generated. This tends to be counterintuitive to users, who consider it a bug or missing feature.¹⁶ There is no straightforward way to derive an oracle for whether something could be generated, so this problem is essentially insoluble without a different approach.

In aid of this, several property-based testing libraries have introduced what is called *integrated shrinking*,¹⁷ where test-case reduction is “bundled” with the generators, so that every generator contains information about how to reduce its generated test cases. In this sense, the internal reduction model we describe in this paper can be thought of as a form of integrated shrinking.

There is however another more widely used implementation of integrated shrinking, the *rose tree*¹⁸ method [4, 5] This approach works by having generators generate a (lazily evaluated) tree consisting of an initial value and possible reductions of it, so that generated values can be reduced by walking the tree. The rose tree method has been implemented in `test.check` (Clojure)¹⁹ and `Hedgehog` (Haskell)²⁰ among others.

Such generators can easily be implemented by pairing a normal random generator of test cases with a test-case reducer, but they save implementation effort by allowing for composition with user defined functions. In particular by supporting the monadic [20] `bind` operator to chain generators together, one can in principle generate anything, as monadic `bind` can be used to express arbitrary computation. Unfortunately in practice the rose tree approach produces poor reductions when `bind` is used,²¹ so generally this approach to integrated shrinking only works well with a relatively restricted set of generators.

Because monadic `bind` can be used to express arbitrary computation, the question of whether internal reduction can work well in these scenarios is essentially equivalent to the question of whether it can work well with arbitrary generators, to which the answer is that it depends. It is certainly possible to construct generators that Hypothesis finds difficult

¹⁶ <https://github.com/typelevel/scalacheck/issues/129>

¹⁷ <https://hypothesis.works/articles/integrated-shrinking/>

¹⁸ A rose tree is a tree where each branch node can have any number of children.

¹⁹ <https://github.com/clojure/test.check>

²⁰ <https://hedgehog.qa/>

²¹ <https://github.com/clojure/test.check/blob/master/doc/growth-and-shrinking.md#unnecessary-bind>

to reduce, but as we saw in Section 4.1 it tends to work well with even large and complex generators written without internal reduction in mind. Also, a key difference is that when Hypothesis has difficulty reducing a generator, typically this is a limitation of its reducer rather than the model: Generally there is some shortlex smaller sequence that would reduce the generated value, but the reducer is unable to find it. Such situations can often be resolved by improving the reducer with no modifications to user code. In contrast, the rose tree model offers no alternative but to add a custom external reducer.

Nevertheless, at present the rose tree model is significantly more widely used than internal reduction. Partly this is just because it predates the internal reduction model, but it is also significantly simpler to implement and easier to understand. Nevertheless, we believe that the benefits of internal reduction are worth the increased implementation complexity, and hope this paper will aid readers' understanding of its model.

6.3 Choice Sequences to Improve Generation

We have introduced the term *choice sequence* to refer to the binary decisions made during random generation, which we use to regard random generators as deterministic parsers of sequences of bits. Similar approaches have been used elsewhere. Most other usage has focused not on test-case reduction but instead on improving the quality of generated test cases by using coverage guided fuzzing. For example, *crowbar* [3] is an OCaml library for providing property-based testing built on top of the AFL Fuzzer²² using this approach. *DeepState* [6] is a unit testing library for C++ which supports either symbolic execution or coverage guided fuzzing. These effectively use a choice sequence, encoded as a sequence of bytes provided by the fuzzer, to make nondeterministic decisions.

Recent work in *Zest* [21] also uses a choice sequence model to improve the quality of generated test case, but uses the term “parameters” to refer to the individual bits. We prefer our “choice sequence” terminology, as the interpretation of a given bit can change during reduction (e.g. a bit that once chose whether to terminate a list might become part of a generated value) so we find thinking of them as parameters a little misleading.

6.4 Other Uses of Internal Reduction

The idea of internal reduction as shortlex optimization originates with Hypothesis, but the idea of manipulating a generator to produce smaller results predates it. The main prior art of which we are aware is *Seq-Reduce* [23], a Csmith mode that attempts to reduce the length of the choice sequence by regenerating parts of it. *Seq-Reduce* was only designed to work with Csmith, and was abandoned due to disappointing results, while we have shown that with internal reduction is both broadly applicable and can work well with Csmith in particular. We have not investigated why we see such a substantial difference between the two approaches, but think it likely that its approach of randomly regenerating parts of the test case was unlikely to work without more structural information such as we describe in Section 3.2.

In addition, there are two significant production implementations of internal reduction that have appeared subsequent to Hypothesis, in both cases explicitly based on its approach. These are *DeepState*, which we also mention in Section 6.3, and *theft*²³, a property-based testing library for C. Both share our approach of internal reduction as shortlex optimization, but have their own reducer implementations.

²² <https://github.com/google/AFL>

²³ <https://github.com/silentbicycle/theft>

7 Conclusion and Future Work

We have presented internal reduction, an approach that performs test-case reduction on generated test cases by manipulating the behavior of the generator that produced them.

The key advantages of internal reduction over conventional, external, reduction are that, by operating solely on the behavior of the generator, it a) provides “free” reduction for arbitrary generators, saving the need to write a new reducer, and b) ensures that reduced test cases are ones that could have been generated, avoiding the test-case validity problem.

As demonstrated by our experimental results, the size of the reduced test cases found by internal reduction is competitive with that found by general purpose reducers such as delta debugging or the reducers typically found in property-based testing libraries, at a moderate increase in reduction cost. Unsurprisingly, there is still a large size gap between its results and those of more specialized reducers such as C-Reduce. We expect that this will continue to be the case, and do not suggest internal reduction as the best model when it is worth investing significant engineering effort in a specialized reducer for a particular test-case format.

Nevertheless, by providing good quality test-case reduction “for free”, internal reduction has significantly improved the user experience of property-based testing in Hypothesis, and the other testing tools we mention in Section 6.4, and is likely to be useful to other users of random generation, especially those not currently using test-case reduction.

Internal test-case reduction has been used in Hypothesis for over four years now, and we consider it a mature and proven technology. Future work on the Hypothesis reducer will seek to improve its performance, and likely will see the development of further reduction passes and heuristics that expand the set of generators it works well for. We’re particularly interested in exploring whether we can improve its performance on larger initial choice sequences, and hope to do further work based on attempting to lift the 8KB buffer size restriction we saw in our experiments with Csmith-generated programs in Section 4.1.

Another exciting line of research is the use of the choice sequence model to implement other functionality. As we discuss in Section 6.3, there are a number of implementations that use this idea to provide coverage-guided fuzzing. Hypothesis has some limited support for this, which we are intending to expand further in future. Additionally, Hypothesis has an implementation of targeted property-based testing [15], which guides generation towards test cases maximizing or minimizing some objective function. The advantages of the choice sequence model for targeted property-based testing are much the same as that for test-case reduction: It provides a fully generic mechanism that requires no user intervention, and ensures that all provided test cases are ones that could have been generated, significantly easing the validity problem. In contrast, prior attempts at fully automating targeted property-based testing (that is, implementing it without requiring user provided mutation functions) in [16] required a great deal of care to ensure valid test cases were produced.

In general, the choice sequence model has proven flexible and powerful, allowing us to implement advanced features with minimal negative impact on users, and without requiring any user expertise in the subject. This makes Hypothesis a powerful tool for creating production implementations of software testing research ideas. We intend to continue using it as such, and encourage other researchers to do the same.

References

- 1 Thomas Arts, John Hughes, Joakim Johansson, and Ulf T. Wiger. Testing telecoms software with quviq quickcheck. In Marc Feeley and Philip W. Trinder, editors, *Proceedings of the 2006 ACM SIGPLAN Workshop on Erlang, Portland, Oregon, USA, September 16, 2006*, pages 2–10. ACM, 2006. doi:10.1145/1159789.1159792.
- 2 Koen Claessen and John Hughes. Quickcheck: a lightweight tool for random testing of haskell programs. In Martin Odersky and Philip Wadler, editors, *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming (ICFP '00), Montreal, Canada, September 18-21, 2000.*, pages 268–279. ACM, 2000. doi:10.1145/351240.351266.
- 3 Stephen Dolan and Mindy Preston. Testing with crowbar. In *Proceedings of the OCaml Users and Developers Workshop*, September 2017. URL: https://ocaml.org/meetings/ocaml/2017/extended-abstract__2017__stephen-dolan_mindy-preston__testing-with-crowbar.pdf.
- 4 Reid Draper. Proposal: free shrinking with quickcheck. <https://mail.haskell.org/pipermail/libraries/2013-November/021674.html>, 2013. Accessed: 2020-05-25.
- 5 Reid Draper. Writing simple-check. <http://reiddraper.com/writing-simple-check/>, 2013. Accessed: 2020-05-25.
- 6 Peter Goodman and Alex Groce. DeepState: Symbolic unit testing for C and C++. In *NDSS Workshop on Binary Analysis Research*, 2018.
- 7 Google. yapf: Yet another python formatter, 2018. URL: <https://github.com/google/yapf>.
- 8 Alex Groce. private correspondence.
- 9 Alex Groce, Josie Holmes, and Kevin Kellar. One test to rule them all. In Tevfik Bultan and Koushik Sen, editors, *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis, Santa Barbara, CA, USA, July 10 - 14, 2017*, pages 1–11. ACM, 2017. doi:10.1145/3092703.3092704.
- 10 Alex Groce and Jervis Pinto. A little language for testing. In Klaus Havelund, Gerard J. Holzmann, and Rajeev Joshi, editors, *NASA Formal Methods - 7th International Symposium, NFM 2015, Pasadena, CA, USA, April 27-29, 2015, Proceedings*, volume 9058 of *Lecture Notes in Computer Science*, pages 204–218. Springer, 2015. doi:10.1007/978-3-319-17524-9_15.
- 11 Alex Groce, Jervis Pinto, Pooria Azimi, and Pranjal Mittal. TSTL: a language and tool for testing (demo). In Michal Young and Tao Xie, editors, *Proceedings of the 2015 International Symposium on Software Testing and Analysis, ISSTA 2015, Baltimore, MD, USA, July 12-17, 2015*, pages 414–417. ACM, 2015. doi:10.1145/2771783.2784769.
- 12 Ralf Hildebrandt and Andreas Zeller. Simplifying failure-inducing input. In Debra J. Richardson and Mary Jean Harold, editors, *Proceedings of the International Symposium on Software Testing and Analysis, ISSTA 2000, Portland, OR, USA, August 21-24, 2000*, pages 135–145. ACM, 2000. doi:10.1145/347324.348938.
- 13 Renáta Hodován and Ákos Kiss. Practical improvements to the minimizing delta debugging algorithm. In Leszek A. Maciaszek, Jorge S. Cardoso, André Ludwig, Marten van Sinderen, and Enrique Cabello, editors, *Proceedings of the 11th International Joint Conference on Software Technologies (ICSOFT 2016) - Volume 1: ICSOFT-EA, Lisbon, Portugal, July 24 - 26, 2016.*, pages 241–248. SciTePress, 2016. doi:10.5220/0005988602410248.
- 14 Ralf Lämmel and Simon L. Peyton Jones. Scrap your boilerplate with class: extensible generic functions. In Olivier Danvy and Benjamin C. Pierce, editors, *Proceedings of the 10th ACM SIGPLAN International Conference on Functional Programming, ICFP 2005, Tallinn, Estonia, September 26-28, 2005*, pages 204–215. ACM, 2005. doi:10.1145/1086365.1086391.
- 15 Andreas Löscher and Konstantinos Sagonas. Targeted property-based testing. In Tevfik Bultan and Koushik Sen, editors, *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis, Santa Barbara, CA, USA, July 10 - 14, 2017*, pages 46–56. ACM, 2017. doi:10.1145/3092703.3092711.
- 16 Andreas Löscher and Konstantinos Sagonas. Automating targeted property-based testing. In *11th IEEE International Conference on Software Testing, Verification and Validation, ICST 2018, Västerås, Sweden, April 9-13, 2018*, pages 70–80. IEEE Computer Society, 2018. doi:10.1109/ICST.2018.00017.

- 17 Lukasz Langa. black: The uncompromising code formatter, 2018. URL: <https://github.com/ambv/black>.
- 18 David Maclver, Zac Hatfield-Dodds, and Many Contributors. Hypothesis: A new approach to property-based testing. *Journal of Open Source Software*, 4(43):1891, November 2019. doi:10.21105/joss.01891.
- 19 Ghassan Mishserghi and Zhendong Su. HDD: hierarchical delta debugging. In Leon J. Osterweil, H. Dieter Rombach, and Mary Lou Soffa, editors, *28th International Conference on Software Engineering (ICSE 2006), Shanghai, China, May 20-28, 2006*, pages 142–151. ACM, 2006. doi:10.1145/1134307.
- 20 Eugenio Moggi. Notions of computation and monads. *Inf. Comput.*, 93(1):55–92, 1991. doi:10.1016/0890-5401(91)90052-4.
- 21 Rohan Padhye, Caroline Lemieux, Koushik Sen, Mike Papadakis, and Yves Le Traon. Semantic fuzzing with zest. In Dongmei Zhang and Anders Møller, editors, *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2019, Beijing, China, July 15-19, 2019*, pages 329–340. ACM, 2019. doi:10.1145/3293882.3330576.
- 22 Lee Pike. Smartcheck: automatic and efficient counterexample reduction and generalization. In Wouter Swierstra, editor, *Proceedings of the 2014 ACM SIGPLAN symposium on Haskell, Gothenburg, Sweden, September 4-5, 2014*, pages 53–64. ACM, 2014. doi:10.1145/2633357.2633365.
- 23 John Regehr, Yang Chen, Pascal Cuoq, Eric Eide, Chucky Ellison, and Xuejun Yang. Test-case reduction for C compiler bugs. In Jan Vitek, Haibo Lin, and Frank Tip, editors, *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '12, Beijing, China - June 11 - 16, 2012*, pages 335–346. ACM, 2012. doi:10.1145/2254064.2254104.
- 24 Wikipedia contributors. Shortlex order, 2020. [Online; accessed 10-January-2020]. URL: https://en.wikipedia.org/wiki/Shortlex_order.
- 25 Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. Finding and understanding bugs in C compilers. In Mary W. Hall and David A. Padua, editors, *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011, San Jose, CA, USA, June 4-8, 2011*, pages 283–294. ACM, 2011. doi:10.1145/1993498.1993532.
- 26 Andreas Zeller and Ralf Hildebrandt. Simplifying and isolating failure-inducing input. *IEEE Trans. Software Eng.*, 28(2):183–200, 2002. doi:10.1109/32.988498.

Model-View-Update-Communicate: Session Types Meet the Elm Architecture

Simon Fowler¹ 

University of Edinburgh, United Kingdom

simon.fowler@glasgow.ac.uk

Abstract

Session types are a type discipline for communication channel endpoints which allow conformance to protocols to be checked statically. Safely implementing session types requires linearity, usually in the form of a linear type system. Unfortunately, linear typing is difficult to integrate with graphical user interfaces (GUIs), and to date most programs using session types are command line applications.

In this paper, we propose the first principled integration of session typing and GUI development by building upon the Model-View-Update (MVU) architecture, pioneered by the Elm programming language. We introduce λ_{MVU} , the first formal model of the MVU architecture, and prove it sound. By extending λ_{MVU} with *commands* as found in Elm, along with *linearity* and *model transitions*, we show the first formal integration of session typing and GUI programming. We implement our approach in the Links web programming language, and show examples including a two-factor authentication workflow and multi-room chat server.

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

Keywords and phrases Session types, concurrent programming, Model-View-Update

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2020.14

Related Version An extended version of the paper is available on arXiv (<https://arxiv.org/abs/1910.11108>).

Supplementary Material ECOOP 2020 Artifact Evaluation approved artifact available at <https://doi.org/10.4230/DARTS.6.2.13>.

Funding This work was supported by ERC Consolidator Grant Skye (grant no. 682315) and an ISCF Metrology Fellowship grant provided by the UK government's Department for Business, Energy and Industrial Strategy (BEIS).

Acknowledgements I thank Jake Browning for sparking my interest in Elm and for his help with an early prototype of the Links MVU library; Sára Decova for a previous version of the multi-room chat server example; Sam Lindley for many useful discussions and suggestions; and James Cheney, April Gonçalves, and the anonymous ECOOP PC and AEC reviewers for detailed comments.

1 Introduction

Modern applications are necessarily concurrent and distributed. Along with concurrency and distribution naturally comes communication, but communication protocols are typically informally described, resulting in costly runtime failures and code maintainability issues.

Session types [23, 24] are a type discipline for communication channel endpoints which allow conformance to a protocol to be checked statically rather than after an application is deployed. Many distributed GUI applications, such as chat applications or multiplayer games, would benefit from session-typed communication with a server. Unfortunately, safely implementing session types requires a linear type system, but safely integrating linear resources and GUIs is nontrivial. As a consequence, to date most programs using session types are batch-style applications run on the command line.

¹ now at University of Glasgow, United Kingdom



© Simon Fowler;

licensed under Creative Commons License CC-BY

34th European Conference on Object-Oriented Programming (ECOOP 2020).

Editors: Robert Hirschfeld and Tobias Pape; Article No. 14; pp. 14:1–14:28

Leibniz International Proceedings in Informatics



LIPICs Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany



14:2 Model-View-Update-Communicate

The lack of a principled integration of GUI applications and session types is a significant barrier to their adoption. In this paper, we bridge this gap by extending the Model-View-Update (MVU) architecture, pioneered by the Elm programming language, to support linear resources. We present λ_{MVU} , a core formalism of the MVU architecture, and an extended version of λ_{MVU} which supports session-typed communication. Informed by the formal development, we provide a practical implementation in the Links programming language [10].

Session types by example. Let us consider a two-factor authentication workflow, introduced by Fowler et al. [20]. A user first enters their credentials. If correct, the server can then either grant access, or send a challenge key. If challenged, the user enters the challenge code into a hardware token, which generates a response to be entered into the web page. The server then either authenticates the user or denies access.

We can describe the two-factor authentication example as a session type as follows:

$$\begin{array}{ll} \text{TwoFactorServer} \triangleq & \text{TwoFactorClient} \triangleq \\ ?(\text{Username}, \text{Password}).\oplus\{ & !(\text{Username}, \text{Password}).\&\{ \\ \quad \text{Authenticated} : \text{ServerBody}, & \quad \text{Authenticated} : \text{ClientBody}, \\ \quad \text{Challenge} : !\text{ChallengeKey}.?\text{Response}. & \quad \text{Challenge} : ?\text{ChallengeKey}!\text{Response}. \\ \quad \oplus\{\text{Authenticated} : \text{ServerBody}, & \quad \&\{\text{Authenticated} : \text{ClientBody}, \\ \quad \quad \text{AccessDenied} : \text{End}\}, & \quad \text{AccessDenied} : \text{End}\}, \\ \quad \text{AccessDenied} : \text{End}\} & \quad \text{AccessDenied} : \text{End}\} \end{array}$$

The `TwoFactorServer` type shows the session type for the server, which firstly receives (?) the credentials from the client, and then chooses (\oplus) whether to authenticate, deny access, or issue a challenge. If the server issues a challenge, it sends (!) the challenge string, awaits the response, and then chooses whether to accept or reject the request. The `ServerBody` type abstracts over the actions performed in the remainder of the application, for example taking out a loan. The `TwoFactorClient` type is the *dual* of the `TwoFactorServer` type: where the server sends, the client receives, and where the client sends, the server receives. The $\&$ construct denotes offering a choice of branches. Suppose we have constructs for sending along, receiving from, and closing an endpoint:

$$\text{send} : (A \times !A.S) \rightarrow S \qquad \text{receive} : ?A.S \rightarrow (A \times S) \qquad \text{close} : \text{End} \rightarrow \mathbf{1}$$

Let us also suppose we have constructs for selecting and offering a choice:

$$\begin{array}{ll} \text{select } \ell_j M : S_j & \text{where } M \text{ has session type } \oplus\{\ell_i : S_i\}_{i \in I}, \text{ and } j \in I \\ \text{offer } M \{\ell_i(x_i) \mapsto N_i\}_{i \in I} : A & \text{where } M \text{ has session type } \&\{\ell_i : S_i\}_{i \in I}, \text{ each } x_i \text{ binds an} \\ & \text{endpoint with session type } S_i, \text{ and each } N_i \text{ has type } A \end{array}$$

We can write a server implementation as follows:

$$\begin{array}{l} \text{twoFactorServer} : \text{TwoFactorServer} \rightarrow \mathbf{1} \\ \text{twoFactorServer}(s) \triangleq \text{let } ((\text{username}, \text{password}), s) = \text{receive } s \text{ in} \\ \quad \text{if } \text{checkDetails}(\text{username}, \text{password}) \text{ then} \\ \quad \quad \text{let } s = \text{select } \text{Authenticated } s \text{ in } \text{serverBody}(s) \\ \quad \text{else let } s = \text{select } \text{AccessDenied } s \text{ in } \text{close } s \end{array}$$

To implement session-typed communication safely, we require a linear type system [44] to ensure each communication endpoint is used exactly once: as an example, without linearity it would be possible to attempt to receive the credentials twice.

Linearity and GUIs. We can also write a client application:

$$\begin{array}{l} \text{twoFactorClient} : (\text{Username} \times \text{Password} \times \text{TwoFactorClient}) \rightarrow \mathbf{1} \\ \text{twoFactorClient}(\text{username}, \text{password}, s) \triangleq \\ \quad \text{let } s = \text{send } ((\text{username}, \text{password}), s) \text{ in} \\ \quad \text{offer } s \{\text{Authenticated}(s) \mapsto \text{clientBody}(s) \\ \quad \quad \text{Challenge}(s) \mapsto \text{let } (\text{key}, s) = \text{receive } s \text{ in} \\ \quad \quad \quad \text{let } s = \text{send } (\text{generateResponse}(\text{key}), s) \text{ in} \\ \quad \quad \quad \text{offer } s \{\text{Authenticated}(s) \mapsto \text{clientBody}(s) \\ \quad \quad \quad \quad \text{AccessDenied}(s) \mapsto \text{close } s; \text{loginFailed}\} \\ \quad \quad \text{AccessDenied}(s) \mapsto \text{close } s; \text{loginFailed}\} \end{array}$$

However, such a client is of little use, as it sends only a pre-defined set of credentials, and the step where a user enters the response to the challenge is replaced by a function `generateResponse`. Ideally, we would like the credentials to be entered into a GUI, and for a button press to trigger the session communication with the server.

Let us attempt to write a GUI for the first stage of the two-factor authentication example; as HTML is well-understood, we concentrate on web pages in the remainder of the paper.

```
render(c)  $\triangleq$ 
  <html>
    <body>
      <input id = "username"></input>
      <input id = "password"></input>
      <button onClick = login(c)>Submit</button>
    </body>
  </html>

login(c)  $\triangleq$   $\lambda().$ 
  let user = getContents("username") in
  let pass = getContents("password") in
  let c = send ((user, pass), c) in
  handleResponse(c)
```

Given a channel c of type `TwoFactorClient`, the `render` function generates a web page with input boxes for the username and password, and a button to submit the credentials. The `login` function, triggered when the button is clicked, retrieves the username and password from the two input boxes, and sends the credentials along c . The `handleResponse` function, which we omit, receives the response from the server and updates the web page.

On first inspection, this implementation seems sound since the endpoint c is used linearly. However, the above attempt is unsound due to the asynchronous nature of GUI programming: there is nothing stopping the user pressing the button twice and sending the credentials twice along c , in contravention of the session type. As a further complication, suppose we augmented the protocol with a “forgotten password” branch, triggered by another button. This would require two instances of c in the GUI, again violating linearity:

```
<button onClick = login(c)>Submit</button>
<button onClick = reset(c)>Reset password</button>
```

It is clear that directly embedding linear resources into a GUI is a non-starter. A more successful approach involves spawning a separate process which contains the linear resource, and which receives *non-linear* messages from the GUI. Upon receiving a GUI message, the process can then perform the session communication, while ignoring duplicate GUI messages:

```
render(c)  $\triangleq$ 
  let pid = spawn handler(c) in
  <html>
    <body>
      <input id = "username"></input>
      <input id = "password"></input>
      <button onClick = login(pid)>Submit</button>
    </body>
  </html>

login(pid)  $\triangleq$   $\lambda().$ 
  let user = getContents("username") in
  let pass = getContents("password") in
  pid! SubmitLogin(user, pass)

handler(c)  $\triangleq$ 
  case (get ()) {
    SubmitLogin(user, pass)  $\mapsto$ 
      let c = send ((user, pass), c) in
      handleResponse(c)
  }
```

The `render` function begins by spawning `handler(c)` as a separate process with an incoming message queue (or *mailbox*), returning the process ID pid . As before, the `login` function is triggered by pressing the button, and retrieves the credentials from the web page. Instead of communicating on the channel directly, it sends a `SubmitLogin` message containing the credentials to the process ID of handler process, written $pid! `SubmitLogin(user, pass)`. The handler process retrieves the message from its mailbox (`get ()`), and can then communicate with the server over the linear endpoint. Such an approach also scales to the “forgotten password” extension, by adding another GUI message.$

The above approach is used by Fowler et al. [20], who provide the first integration of session types and web application development, including the ability to gracefully handle failures such as the user closing their browser mid-session. Unfortunately, the approach is

14:4 Model-View-Update-Communicate

brittle and ad-hoc. All interaction with the web page occurs using imperative operations such as `getContents` and `setContents`; contrary to best practices such as the Model-View-Controller (MVC) [30] pattern, the state of the web page is not derived directly from the data contained by the application. Furthermore, there is no connection between the state of the handler process and what is displayed on the web page: this can easily lead to mismatches between the possible GUI messages which can be sent and which can be handled.

Model-View-Update. This paper is about doing better. Our approach is to formalise Model-View-Update, an architectural pattern for GUI development popularised by the Elm programming language [1], and extend it to support linear resources. MVU is an appealing starting point as it is particularly suited to functional programming. Furthermore, MVU has directly inspired popular technologies such as Redux [5] and the Flux architecture [4], which are used with the popular React [2] frontend web framework for JavaScript.

The Elm programming language [1] is a functional programming language designed for writing web applications. Elm was originally designed to use *functional reactive programming* (FRP) [14], where time-varying *signals* can be used to construct reactive web applications. A paper describing Elm, and its core formal semantics, was published at PLDI 2013 [12].

For many languages, that would be the end of the story. But unusually for a research language, Elm gained a user community, and a standard architectural pattern known as *The Elm Architecture* grew organically to such a point that Elm abandoned FRP altogether [11]. At its core, The Elm Architecture is a descendant of MVC where a *model* contains the state of the application; a *view function* renders the model; and the rendered model produces *messages* which are handled by an *update* function to produce a new model. More generally, this pattern has been referred to as *Model-View-Update*, or MVU for short [3, 40].

Consider the following web application, where a user enters text into a text box, and the application displays the text, reversed:

```
hello
olleh
```

We can write this example using MVU as follows:

```
Model ≜ (contents : String)
Message ≜ UpdateBox(String)

model : Model
model ≜ (contents = "")

update : (Message × Model) → Model
update ≜ λ(UpdateBox(str), m).(contents = str)

view : Model → Html(Message)
view ≜ λmodel.html
  <input type = "text" value = {model.contents}
    onInput = {λstr.UpdateBox(str)}></input>
  <div>
    {htmlText (reverseString (model.contents))}
  </div>

(model, view, update)
```

We define two type aliases: the `Model` captures the state of the application and is defined as a record with a single `String` field, `contents`. `Messages` are produced as a result of user interaction. The `Message` type is defined as a singleton variant type with constructor `UpdateBox`, containing the updated value of the text box.

The view function renders a model. It has the type `Model → Html(Message)`, which is a function taking a `Model` as its argument, and returning HTML which may produce messages of type `Message`. The `value = {model.contents}` attribute of the `input` box states that the contents of the text box should reflect the `contents` field of the model. The `onInput` attribute is an *event handler*: its body is a function taking the current value of the input box (`str`) and producing an `UpdateBox` message containing the updated contents of the box. The contents of the `div` tag are derived from the reversed contents.

Syntax		
Types	$A, B, C ::= \mathbf{1} \mid A \rightarrow B \mid A \times B \mid A + B \mid \mathbf{String} \mid \mathbf{Int}$ $\mid \mathbf{Html}(A) \mid \mathbf{Attr}(A)$	
String literals	s	
Integers	n	
Terms	$L, M, N ::= x \mid \lambda x.M \mid \mathbf{rec} f(x). M \mid M N \mid () \mid s \mid n$ $\mid (M, N) \mid \mathbf{let} (x, y) = M \mathbf{in} N$ $\mid \mathbf{inl} x \mid \mathbf{inr} x \mid \mathbf{case} L \{ \mathbf{inl} x \mapsto M; \mathbf{inr} y \mapsto N \}$ $\mid \mathbf{htmlTag} t M N \mid \mathbf{htmlText} M \mid \mathbf{htmlEmpty}$ $\mid \mathbf{attr} ak M \mid \mathbf{attrEmpty} \mid M \star N$	
Tag names	t	
Attribute names	at	
Attribute keys	$ak ::= at \mid h$	
Event handler names	h	
Typing rules for terms $\Gamma \vdash M : A$		
$\frac{\mathbf{T-HtmlTAG} \quad \Gamma \vdash M : \mathbf{Attr}(A) \quad \Gamma \vdash N : \mathbf{Html}(A)}{\Gamma \vdash \mathbf{htmlTag} t M N : \mathbf{Html}(A)}$	$\frac{\mathbf{T-HtmlTEXT} \quad \Gamma \vdash M : \mathbf{String}}{\Gamma \vdash \mathbf{htmlText} M : \mathbf{Html}(A)}$	$\frac{\mathbf{T-HtmlEMPTY}}{\Gamma \vdash \mathbf{htmlEmpty} : \mathbf{Html}(A)}$
$\frac{\mathbf{T-Attr} \quad \Gamma \vdash M : \mathbf{String}}{\Gamma \vdash \mathbf{attr} at M : \mathbf{Attr}(A)}$	$\frac{\mathbf{T-EvtAttr} \quad \Gamma \vdash M : \mathbf{ty}(h) \rightarrow A}{\Gamma \vdash \mathbf{attr} h M : \mathbf{Attr}(A)}$	$\frac{\mathbf{T-AttrEmpty}}{\Gamma \vdash \mathbf{attrEmpty} : \mathbf{Attr}(A)}$
$\frac{\mathbf{T-HtmlAppend} \quad \Gamma \vdash M : \mathbf{Html}(A) \quad \Gamma \vdash N : \mathbf{Html}(A)}{\Gamma \vdash M \star N : \mathbf{Html}(A)}$	$\frac{\mathbf{T-AttrAppend} \quad \Gamma \vdash M : \mathbf{Attr}(A) \quad \Gamma \vdash N : \mathbf{Attr}(A)}{\Gamma \vdash M \star N : \mathbf{Attr}(A)}$	

■ **Figure 1** Syntax and typing rules for λ_{MVU} terms.

The `update` function takes a message and previous model as its arguments, and produces a new model. In this case, the `update` function constructs a new model where the `contents` field is set to the payload of the `UpdateBox` message. Finally, the program is a 3-tuple containing the initial model, and the view and update functions.

To achieve our goal of a formal integration of session typing and GUI programming, we first formalise MVU, and then generalise the architecture to support linear models and messages. Supporting linearity poses some challenges, as we will see in §3.

1.1 Contributions

The overarching contribution of this paper is the first principled integration of session-typed communication with a GUI framework. Concretely, we make three contributions:

1. We introduce the first formal model of the MVU architecture, λ_{MVU} (§2). We prove (§2.3) that λ_{MVU} satisfies preservation and event progress properties.
2. We extend λ_{MVU} with *commands*, *linearity*, and *model transitions* (§3), which allow λ_{MVU} to support GUIs incorporating session-typed communication, and we prove the soundness of the extended calculus.
3. We implement the architecture in the Links web programming language. We show an extended example of a chat application where client code uses the linear MVU framework, and where client-server communication happens over session-typed channels (§4).

The implementation and examples are available in the paper’s companion artifact.

Event name ev	Event Handler h ($handler(ev)$)	Payload type ($ty(ev)$, $ty(h)$)	Payload Description
click	onClick	1	Unit value
input	onInput	String	Updated contents of a text field
keyUp	onKeyUp	Int	Key code
keyDown	onKeyDown	Int	Key code

■ **Figure 2** Example event signatures.

2 Model-View-Update, Formally

In this section, we formalise MVU as a core calculus, λ_{MVU} , an extension of the simply-typed λ -calculus with products, sums, HTML, and event handling. Even without extensions, λ_{MVU} is expressive enough to support many common applications such as form handling.

2.1 Syntax

Types. Figure 1 shows the syntax and typing rules for λ_{MVU} . Types are ranged over by A, B, C , and consist of the unit type **1**, functions $A \rightarrow B$, products $A \times B$, sums $A + B$, and string and integer types. Types $\text{Html}(A)$ and $\text{Attr}(A)$ are the type of HTML elements and attributes which can produce messages of type A .

Terms. Terms, ranged over by L, M, N , include variables, λ abstractions, anonymous recursive functions, function application, the unit value, string literals, integers, and sum and pair introduction and elimination. The remaining terms encode HTML *elements* and *attributes*. The **htmlTag** $t M N$ construct denotes an HTML element with tag name t (for example, `div`), attributes M , and children N ; the **htmlText** M construct describes a text node with text M ; and **htmlEmpty** defines an empty HTML node.

The **attr** $ak M$ construct describes an attribute with key ak and body M , where the key ak is either an attribute name at or an event handler name h . The **attrEmpty** construct defines an empty attribute.

The $M \star N$ operator appends two HTML elements or attributes. Since both HTML elements and attributes support a unit element (**htmlEmpty** and **attrEmpty** respectively), elements and attributes together with \star form two monoids.

Events. We model interaction with the Document Object Model (DOM) through *events*, which model those dispatched by a browser. An *event signature* is a 3-tuple (ev, h, A) consisting of an event name ev , handler name h , and payload type A . We require a bijective mapping between event and handler names. Figure 2 describes example event signatures used in the remainder of the paper. We consider four primitive events: `click`, which is fired when an element is clicked; `input`, which is fired when the contents of a text field are changed; and `keyUp` and `keyDown`, which are fired when a key is pressed while focused on an element.

Event handlers are attached to elements as attributes, and generate a message in response to an event. We write $handler(ev)$ to refer to the handler for ev : for example, $handler(click) = \text{onClick}$. We write $ty(ev)$ to refer to the payload type of ev and write $ty(h)$ for the payload type of an event handled by h . As an example, both $ty(click) = \mathbf{1}$ and $ty(\text{onClick}) = \mathbf{1}$.

Term typing. Term typing rules for λ -calculus constructs are standard, so are omitted. Rule T-HTMLTAG states that **htmlTag** $t M N$ can be given type $\text{Html}(A)$ if its attributes M have type $\text{Attr}(A)$ and children have type $\text{Html}(A)$. Text nodes **htmlText** M do not produce any messages, and so have type $\text{Html}(A)$ if M has type **String** (T-HTMLTEXT); similarly, **htmlEmpty** has type $\text{Html}(A)$ (T-HTMLEMPTY).

Values	$U, V, W ::= \lambda x.M \mid \mathbf{rec} f(x).M \mid () \mid (V, W) \mid \mathbf{inl} V \mid \mathbf{inr} V \mid s \mid n$ $\mid \mathbf{htmlTag} \ t \ V \ W \mid \mathbf{htmlEmpty} \mid \mathbf{htmlText} V$ $\mid \mathbf{attr} \ ak \ V \mid \mathbf{attrEmpty} \mid V \star W$
Events	$e ::= \mathbf{ev}(V)$
DOM Pages	$D ::= \mathbf{domTag}(\vec{e}) \ t \ V \ D \mid \mathbf{domText} V \mid \mathbf{domEmpty} \mid D \star D'$
Active thread	$T ::= \mathbf{idle} \ V_m \mid M$
Function state	$F ::= (V_v, V_u)$
Processes	$P, Q ::= \mathbf{run} M \mid \langle T \mid F \rangle \mid ((M)) \mid P \parallel Q$
Configurations	$\mathcal{C} ::= P \S D$
Process contexts	$\mathcal{P} ::= [] \mid \mathcal{P} \parallel P$
DOM contexts	$\mathcal{D} ::= [] \mid \mathbf{domTag}(\vec{e}) \ t \ V \ \mathcal{D} \mid \mathcal{D} \star D \mid D \star \mathcal{D}$
Thread contexts	$\mathcal{T} ::= \mathbf{run} E \mid \langle E \mid F \rangle \mid ((E))$

■ **Figure 3** Runtime syntax for λ_{MVU} .

Rule T-ATTR assigns attributes **attr** at M type $\mathbf{Attr}(A)$ for any A if M has type \mathbf{String} . Rule T-EVTATTR types event handler attributes **attr** $h M$: if the event handler M has type $\mathbf{ty}(h) \rightarrow A$ (i.e., it *produces messages of type* A), then the attribute can be given type $\mathbf{Attr}(A)$. Finally, T-ATTREMPTY states that the empty attribute **attrEmpty** has type $\mathbf{Attr}(A)$ for any type A . We overload the \star operator to append both HTML elements and attributes (T-HTMLAPPEND and T-ATTRAPPEND).

Syntactic sugar. We assume the usual encodings of records as pairs and variant types as binary sums, and use pattern matching notation. It is useful to be able to write HTML using XML-like notation, where an *antiquoted expression* $\{M\}$ allows a term M to be embedded within an HTML tree. The view function from the introduction desugars to:

$$\lambda model.$$

$$(\mathbf{htmlTag} \ \text{input}$$

$$((\mathbf{attr} \ \text{type} \ \text{"text"}) \star (\mathbf{attr} \ \text{value} \ model.\text{contents}) \star$$

$$(\mathbf{attr} \ \text{onInput} \ (\lambda str.\text{UpdateBox}(str)))) \ \mathbf{htmlEmpty}) \star$$

$$\mathbf{htmlTag} \ \text{div} \ \mathbf{attrEmpty} \ (\mathbf{htmlText} \ \text{reverseString} \ (model.\text{contents}))$$

The formal definitions and desugaring translations are unsurprising; the details can be found in the extended version [18].

2.2 Operational Semantics

We can now provide λ_{MVU} with a small-step operational semantics.

2.2.1 Runtime Syntax

Figure 3 describes the runtime syntax of λ_{MVU} . Values, ranged over by U, V, W , are standard. An event $\mathbf{ev}(V)$ consists of event name \mathbf{ev} and payload V . We write ϵ for an empty meta-level sequence, and use \cdot for sequence concatenation. DOM pages, ranged over by D , are the runtime representation of HTML, where tags $\mathbf{domTag}(\vec{e}) \ t \ V \ D$ contain an event queue \vec{e} of events dispatched to the element.

Concurrency. Concurrency is vital when modelling GUI applications as event handling is asynchronous: computation triggered by a user interaction should not block the UI. Concurrency is also essential when considering session-typed communication. We therefore formulate the calculus as a concurrent λ -calculus in the style of Niehren et al. [36], by augmenting the simply-typed λ -calculus with processes and concurrent reduction.

Meta-level definitions		$\text{handlers}(\text{ev}, \mathbf{attrEmpty}) = \epsilon$ $\text{handlers}(\text{ev}, V \star W) = \text{handlers}(\text{ev}, V) \cdot \text{handlers}(\text{ev}, W)$ $\text{handlers}(\text{ev}, \mathbf{attr at } V) = \epsilon$ $\text{handlers}(\text{ev}, \mathbf{attr } h V) = \begin{cases} V & \text{if handler}(\text{ev}) = h \\ \epsilon & \text{otherwise} \end{cases}$
$\text{handle}(m, (v, u), \text{msg}) \triangleq$ $\mathbf{let } m' = u(\text{msg}, m) \mathbf{ in}$ $(m', v m')$		
Process reduction		$P \longrightarrow P'$
EP-HANDLE	$\langle \mathbf{idle } V_m \mid F \rangle \parallel ((V)) \longrightarrow \langle \text{handle}(V_m, F, V) \mid F \rangle$	
EP-PAR	$P_1 \parallel P_2 \longrightarrow P'_1 \parallel P_2 \quad \text{if } P_1 \longrightarrow P'_1$	
EP-LIFTT	$\mathcal{T}[M] \longrightarrow \mathcal{T}[N] \quad \text{if } M \longrightarrow_M N$	
Configuration reduction		$\mathcal{C} \longrightarrow \mathcal{C}'$
E-RUN	$\mathcal{P}[\mathbf{run}(V_m, V_v, V_u)] \S D \longrightarrow \mathcal{P}[\langle (V_m, V_v V_m) \mid (V_v, V_u) \rangle] \S D$	
E-UPDATE	$\mathcal{P}[\langle (V_m, U) \mid F \rangle] \S D \longrightarrow \mathcal{P}[\langle \mathbf{idle } V_m \mid F \rangle] \S D' \quad \text{where } \text{diff}(U, D) = D'$	
E-INTERACT	$P \S \mathcal{D}[\mathbf{domTag}(\vec{e}) \text{ t } U D] \longrightarrow P \S \mathcal{D}[\mathbf{domTag}(\vec{e} \cdot \text{ev}(V)) \text{ t } U D]$ for some ev, V such that $\vdash \text{ev}(V)$	
E-EVT	$P \S \mathcal{D}[\mathbf{domTag}(\text{ev}(W) \cdot \vec{e}) \text{ t } U D] \longrightarrow P \parallel ((V_1 W)) \parallel \dots \parallel ((V_n W)) \S \mathcal{D}[\mathbf{domTag}(\vec{e}) \text{ t } U D]$ where $\text{handlers}(\text{ev}, U) = \vec{V}$	
E-STRUCT	$\mathcal{C} \longrightarrow \mathcal{C}' \quad \text{if } \mathcal{C} \equiv \mathcal{C}_1, \mathcal{C}_1 \longrightarrow \mathcal{C}_2, \text{ and } \mathcal{C}_2 \equiv \mathcal{C}'$	
E-LIFTP	$P \S D \longrightarrow P' \S D \quad \text{if } P \longrightarrow P'$	

■ **Figure 4** Reduction rules for λ_{MVU} terms and configurations.

Processes. An *initialisation process* $\mathbf{run } M$ evaluates the initial system state written by a user, where M is a 3-tuple containing the initial model, view function, and update function. An *event loop process* $\langle T \mid F \rangle$ consists of an active thread T and function state F comprising the view and update functions. The thread can either be $\mathbf{idle } V_m$, meaning the process has current model V_m and is waiting for another message to process, or evaluating a term M . An *event handler process* $((M))$ is spawned to generate a message in response to an event.

Configurations. Concurrent and event-driven reduction happens in the context of a *system configuration* $P \S D$, where P is the concurrent fragment of the system and D is the current DOM page. An MVU program as written by a user is a term M specifying the initial model, view function, and update function, of type $(A \times (A \rightarrow \text{Html}(B)) \times ((B \times A) \rightarrow A))$. A program is evaluated in the context of an *initial configuration*:

► **Definition 1** (Initial configuration). *An initial configuration for a term M is of the form $\mathbf{run } M \S \mathbf{domEmpty}$.*

Evaluation contexts. Term evaluation contexts E (omitted) are set up for call-by-value, left-to-right evaluation. Process contexts \mathcal{P} allow reduction under parallel composition. Thread contexts \mathcal{T} allow reduction inside threads. DOM contexts \mathcal{D} allow us to focus on each element of a DOM forest; note that they deliberately allow non-unique decomposition in order to support nondeterministic reduction.

2.2.2 Reduction Rules

Figure 4 shows the reduction rules for λ_{MVU} processes and configurations; reduction on terms is standard β -reduction. Reduction on configurations is defined modulo the associativity and commutativity of parallel composition.

Diffing. As DOM pages include event queues, they contain strictly more information than HTML. To avoid losing pending events, we require a diffing operation. Define $\text{erase}(D)$ as the operation $\text{erase}(\text{domTag}(\vec{e}) \text{ t } U \ D) = \text{htmlTag} \text{ t } U \ (\text{erase}(D))$, with the other cases defined recursively. DOM pages can be modified by adding a node with an empty queue, removing a node, or updating a node's attributes. We define operation $\text{diff}(U, D) = D'$ if $\text{erase}(D') = U$, and D' is obtained from D by the minimum number of insertions and deletions.

Semantics by example. Let us return to our original example from §1: a box and a text node displaying the reversed box contents. We reuse the `view` and `update` functions and let $V_m = (\text{contents} = "")$, $V_v = \text{view}$, and $V_u = \text{update}$. We extend the HTML syntactic sugar to pages, letting $\llbracket - \rrbracket$ be a desugaring function and $\llbracket \langle \text{t } \vec{a} \ @ \ \vec{e} \rangle \overrightarrow{D_H} \langle / \text{t} \rangle \rrbracket = \text{domTag}(\vec{e}) \text{ t } \llbracket \vec{a} \rrbracket \llbracket \overrightarrow{D_H} \rrbracket$.

We write \mathcal{R}^+ for the transitive closure of a relation \mathcal{R} . We begin by supplying the model, view, and update parameters to an initial configuration. By E-RUN, we get an event loop process, and then term $V_v \ V_m$ reduces to the initial rendered HTML. By diffing against the empty page, we display the initial DOM page (E-UPDATE).

```

run (V_m, V_v, V_u) § domEmpty
→ (E-RUN)  ((V_m, V_v \ V_m) | (V_v, V_u)) § domEmpty
→+M      <input type = "text" value = ""
           ((V_m,   onInput = {λstr.UpdateBox(str)})></input> ) | (V_v, V_u)) § domEmpty
           <div></div>
→ (E-UPDATE)
           <input type = "text" value = ""
           (idle V_m | (V_v, V_u)) §   onInput = {λstr.UpdateBox(str)} @ ε></input>
           <div @ ε></div>

```

The system now does not reduce until a user interacts with the text box and presses the `k` key, modelled by E-INTERACT. At this point, the event queue for the `input` box receives four events: `click`, `keyDown`, `keyUp`, and `input`, which are processed by rule E-EVT. The `input` element does not have handlers for the `click`, `keyDown`, and `keyUp` events, so no processes are spawned, but *does* contain an `onInput` handler, which handles the `input` event by spawning $((\text{UpdateBox}(\text{"k"})))$.

```

→+ (E-INTERACT)  <input type = "text" value = ""
                  onInput = {λstr.UpdateBox(str)} @ click().
                  (idle V_m | (V_v, V_u)) §   keyDown(75) · keyUp(75) · input("k")></input>
                  <div @ ε></div>
→+ (E-EVT)       <input type = "text" value = ""
                  onInput = {λstr.UpdateBox(str)} @ input("k")>
                  (idle V_m | (V_v, V_u)) §   </input>
                  <div @ ε></div>
→ (E-EVT)        <input type = "text" value = ""
                  onInput = {λstr.UpdateBox(str)}
                  (idle V_m | (V_v, V_u)) || ((UpdateBox("k"))) § @ ε></input>
                  <div @ ε></div>

```

Since $\text{UpdateBox}(\text{"k"})$ is already a value and the event loop process is idle, we can process the message (E-HANDLE). The `handle` meta-function calculates a new model m' by applying the `update` function to a pair of the previous model and the message, calculates a new HTML value v' by applying the `view` function to m' , and returns the pair (m', v') . Finally, the page is diffed against the previous DOM page to generate a new DOM page D' , and the event loop process reverts to being idle:

Typing rules for events $\boxed{\vdash e}$	Typing rules for active threads	$\boxed{\vdash T : \text{EvtLoop}(A, B)}$
$\frac{\text{TE-EVT}}{\cdot \vdash V : \text{ty}(\text{ev})} \quad \vdash \text{ev}(V)$	$\frac{\text{TS-IDLE}}{\cdot \vdash V_m : A} \quad \vdash \text{idle } V_m : \text{EvtLoop}(A, B)$	$\frac{\text{TS-PROCESSING}}{\cdot \vdash M : (A \times \text{Html}(B))} \quad \vdash M : \text{EvtLoop}(A, B)$
Typing rules for processes and configurations		$\boxed{\vdash^\phi P : A} \quad \boxed{\vdash \mathcal{C}}$
$\frac{\text{TP-RUN}}{\cdot \vdash M : (A \times (A \rightarrow \text{Html}(B)) \times ((B \times A) \rightarrow A))} \quad \vdash^\bullet \text{run } M : B$	$\frac{\text{TP-EVENTLOOP}}{\cdot \vdash V_v : A \rightarrow \text{Html}(B) \quad \cdot \vdash V_u : (B \times A) \rightarrow A} \quad \vdash^\bullet \langle T \mid (V_v, V_u) \rangle : B$	
$\frac{\text{TP-THREAD}}{\cdot \vdash M : A} \quad \vdash^\circ ((M)) : A$	$\frac{\text{TP-PAR}}{\vdash^{\phi_1} P_1 : A \quad \vdash^{\phi_2} P_2 : A} \quad \vdash^{\phi_1 + \phi_2} P_1 \parallel P_2 : A$	$\frac{\text{TC-SYSTEM}}{\vdash^\bullet P : A \quad \vdash D : \text{Page}(A)} \quad \vdash P \S D$
Combination of flags		$\boxed{\phi_1 + \phi_2}$
$\circ + \circ = \circ$	$\circ + \bullet = \bullet$	$\bullet + \bullet = \text{undefined}$

■ **Figure 5** Runtime typing for λ_{MVU} .

\rightarrow (EP-HANDLE)	$\langle \text{handle}(V_m, (V_v, V_u), \text{UpdateBox}(\text{"k"})) \mid (V_v, V_u) \rangle \S$	<pre><input type = "text" value = "" onInput = {λstr.UpdateBox(str)} @ε></input> <div @ε></div></pre>
\rightarrow_M^+	$((\text{contents} = \text{"k"}, \langle \langle \text{onInput} = \{\lambda \text{str.UpdateBox}(str)\} \rangle \rangle \mid (V_v, V_u)) \S$	<pre><input type = "text" value = "" onInput = {λstr.UpdateBox(str)} @ε></input> <div @ε></div></pre>
\rightarrow (E-UPDATE)	$\langle \text{idle}(\text{contents} = \text{"k"}) \mid (V_v, V_u) \rangle \S$	<pre><input type = "text" value = "k" onInput = {λstr.UpdateBox(str)} @ε></input> <div @ε>k</div></pre>

2.3 Metatheory

Runtime typing. To reason about the metatheory, we require runtime typing rules, shown in Figure 5. Judgement $\vdash e$ states that the payload of an event e has the payload type specified by its signature. Judgement $\vdash T : \text{EvtLoop}(A, B)$ can be read “Active thread T has model type A and message type B ”. An idle thread **idle** V_m has type $\text{EvtLoop}(A, B)$ if V_m has type A (TS-IDLE). An active thread M currently processing a message has type $\text{EvtLoop}(A, B)$ if M has type $(A \times \text{Html}(B))$, i.e., computes a pair of a new model with type A and HTML which produces messages of type B (TS-PROCESSING).

Judgement $\vdash^\phi P : A$ states that process P is well typed and produces or consumes messages of type A . The parallel composition of two processes $P_1 \parallel P_2$ has message type A if both P_1 and P_2 have message type A (TP-PAR). An event handler process $((M))$ has message type A if term M has type A (TP-THREAD).

An initialisation process **run** M is well-typed if M is a product type where each component has the correct model, view, and update types. An event loop process $\langle T \mid (V_v, V_u) \rangle$ has message type B if its active thread T has model type A and message type B ; its view function

V_v has type $A \rightarrow \text{Html}(B)$; and its update function has type $(B \times A) \rightarrow A$ (TP-EVENTLOOP). Thread flags ϕ ensure that there is precisely one initialisation process or event loop process in a process typeable under flag \bullet .

Judgement $\vdash \mathcal{C}$ states that configuration \mathcal{C} is well-typed: a system configuration $P \ ; \ D$ is well-typed if process P has precisely one event loop process with message type A and page D has type $\text{Page}(A)$. The omitted typing rules for pages (of shape $\vdash D : \text{Page}(A)$) are similar to those for terms of type $\text{Html}(A)$.

Note that we consider only closed configurations and processes since it makes little sense for DOM pages D to contain free variables, and because processes do not bind variables.

We are now well-placed to state some formal results. We omit proofs in the main body of the paper; full proofs can be found in the extended version [18].

Preservation. Reduction preserves typing.

► **Theorem 2** (Preservation). *If $\vdash \mathcal{C}$ and $\mathcal{C} \longrightarrow \mathcal{C}'$, then $\vdash \mathcal{C}'$.*

Progress. The system vacuously satisfies a progress property as it can always reduce by E-INTERACT due to user input. It is more interesting to consider the *event progress* property enjoyed by the system *without* E-INTERACT: either there are no events to process and the system is idle, or the system can reduce. Functional reduction satisfies progress.

► **Lemma 3** (Progress (Terms)). *If $\cdot \vdash M : A$, then either M is a value, or there exists some N such that $M \longrightarrow_M N$.*

Let \longrightarrow_E be the relation \longrightarrow without rule E-INTERACT. The concurrent fragment of the language will reduce until all event handler threads have finished evaluating, and there are no more messages to process. By appeal to Lemma 3, we can show event progress.

► **Theorem 4** (Event Progress). *If $\vdash \mathcal{C}$, either:*

1. *there exists some \mathcal{C}' such that $\mathcal{C} \longrightarrow_E \mathcal{C}'$; or*
2. *$\mathcal{C} = \langle \text{idle } V_m \mid (V_v, V_u) \rangle \ ; \ D$ where D cannot be written $\mathcal{D}[\text{domTag}(\vec{e}) \ t \ V \ W]$ for some non-empty \vec{e} .*

3 λ_{MVU} with Session Types

In this section, we extend λ_{MVU} to support session types. We require three extensions: *commands*, to perform side-effects; *linearity*, to implement session types safely; and *transitions*, to allow multiple model and message types. We begin by showing each extension by example, and show the extended formalism in §3.4.

3.1 Commands

Real-world applications require side-effects. To this end, Elm supports *commands* which describe side-effects to be performed in the event loop. Although commands in Elm are more general, for our purposes, it is particularly useful to be able to spawn a process which will run concurrently and eventually return a message. As an example, we may want to await the result of an expensive computation, and display the result when the computation completes. Letting $\text{naiveFib}(x)$ be the naïve Fibonacci function and assuming an intToString function, we can write:

14:12 Model-View-Update-Communicate

```

Model  $\triangleq$  Maybe(Int)    Message  $\triangleq$  StartComputation | Result(Int)
view : Model  $\rightarrow$  Html(Message)
view =  $\lambda model.$  html
  <html>
    <body>
      { case model {
        Just(result)  $\mapsto$  htmlText intToString(result);
        Nothing  $\mapsto$  htmlText "Waiting ..." } }
      <button onClick = { $\lambda()$ .StartComputation}>Start!</button>
    </body>
  </html>

update : (Message  $\times$  Model)  $\rightarrow$  (Model, Cmd(Message))
update =  $\lambda(message, model).$ 
  case message {
    StartComputation  $\mapsto$  (Nothing, cmdSpawn Result(naiveFib(1000)))
    Result(x)  $\mapsto$  (Just(x), cmdEmpty)
  }

```

The model is of type `Maybe(Int)`, with value `Just(V)` for some integer value V if the result has been computed, or `Nothing` if the application is awaiting the result. The `Message` type is a variant type consisting of `StartComputation` which is sent to start the computation, and `Result(Int)`, which is sent to return a result. The view function renders either the result, or "Waiting..." if no result is available.

The type of the `update` function is changed to return a *pair* of an updated model and a command. In our case, the `StartComputation` message results in a pair of `Nothing` and `cmdSpawn Result(naiveFib(1000))`, which spawns `Result(naiveFib(1000))` to evaluate in a separate thread. When the function (eventually) completes, the thread will have evaluated to a `Result` message, which can be processed by the `update` function to update the model and display the result.

3.2 Linearity

As we showed in §1, safely implementing session types requires linearity: we therefore require linear model and message types. Linearity would also prove useful for other linear resources such as functional arrays with in-place update [44]. Unfortunately, λ_{MVU} as defined so far does not support linearity. Consider `handle`:

```

handle(m, (v, u), msg)  $\triangleq$  let m' = u (msg, m) in (m', v m')

```

The updated model, m' , is used non-linearly as it is returned for use in subsequent requests, and also used to render the model as HTML.

Extraction. Linear resources are needed only when *updating* the model – not when rendering the webpage – as we will not need to communicate on session channels when rendering. If the developer implements a function:

```

extract : A  $\rightarrow$  (A  $\times$  B)

```

where A is the type of a model, and B is the *unrestricted* fragment of the model, we can restore linear usage of the model (letting e be the extraction function):

```

handle(m, (v, u, e), msg)  $\triangleq$  let m' = u (msg, m) in
  let (m', unrM) = e m' in (m', v unrM)

```

An alternative approach would be to assign the view function type $A \rightarrow (A \times \text{Html}(B))$, returning the linear model and allowing it to be re-bound. We would need to modify `handle`:

```

handle(m, (v, u), msg)  $\triangleq$  let m' = u (m, msg) in v m'

```

```

Model  $\triangleq$  (Bool  $\times$  Chan(Ping)  $\times$  Chan(Pong))
view : Model  $\rightarrow$  Html(Message)
view  $\triangleq$   $\lambda$ (pinging, _, _).
  let attr =
    if pinging then
      attrEmpty
    else
      attr "disabled" "true" in
  html
  <html>
  <body>
  <button {attr} onClick = { $\lambda$ () . Click}>
    Send Ping!
  </button>
  </body>
  </html>

Message  $\triangleq$  Click | Ponged
update : (Message  $\times$  Model)  $\rightarrow$  Model
update  $\triangleq$   $\lambda$ (msg, (_, pingCh, pongCh)).
  case msg {
  Click  $\mapsto$ 
    let cmd =
      cmdSpawn ( send (Ping, pingCh);
                  let Pong = receive pongCh in
                  Ponged) in
    ((false, pingCh, pongCh), cmd)
  Ponged  $\mapsto$  ((true, pingCh, pongCh), cmdEmpty)
  }

server : (Chan(Ping)  $\times$  Chan(Pong))  $\rightarrow$  (1  $\rightarrow$  A)
server  $\triangleq$   $\lambda$ (pingCh, pongCh).
  (rec f()).
  let Ping = receive pingCh in
  send (Pong, pongCh); f ()

```

■ **Figure 6** PingPong application using simply-typed channels.

A key disadvantage of this approach is that rendering is no longer a read-only operation, breaking an important abstraction barrier.

Example. We can now write our first session-typed λ_{MVU} application. Our web client consists of a button which, when clicked, triggers the sending of a Ping message to the server. Once clicked, the button is disabled. The server then receives the Ping message and responds with a Pong message; upon receiving the response, the client then re-enables the button.



Simply-typed channels. Before considering a session-typed version of the application, it is instructive to consider a version *without* session typing, shown in Figure 6. Let $\text{Chan}(A)$ be the type of a simply-typed channel over which one can send and receive values of type A . The model is a 3-tuple containing a Boolean value which is true when waiting for the user to click the “Send Ping!” button, and false when waiting for a response; a channel for Ping messages; and a channel for Pong messages. There are two types of UI message: Click denotes that the button has been clicked, and Ponged denotes that a Pong message has been received along the Pong channel.

The view function displays the page, adding the `disabled` attribute to the button if we are waiting for a Pong message. The `update` function case-splits on the UI message: in the case of a Click message raised by the button, the model is updated to set the `pinging` flag to false, and the function creates a command to send a Ping message along `pingCh`, receive a Pong message from `pongCh`, and return a Ponged UI message. In the case of a Ponged message, the model is updated to set the `pinging` flag to true, enabling the button again. The server function models a server thread, which repeatedly receives Ping messages from `pingCh` and sends Pong messages to `pongCh`.

Even in this simple example, it is very easy to communicate incorrectly: if the client neglected to send a Ping message before trying to receiving a Pong message along `pongCh`, then the command would hang forever and the GUI would never re-enable the button. A similar situation would arise if the server received the Ping message but failed to respond.

```

PingPong  $\triangleq$   $\mu t. !\text{Ping}. ?\text{Pong}. t$ 
UModel  $\triangleq$  UPinging | UWaiting

Model  $\triangleq$  Pinging(PingPong) | Waiting
Message  $\triangleq$  Click | Ponged(PingPong)

view : UModel  $\rightarrow$  Html(Message)
view  $\triangleq$   $\lambda uModel.$ 
  let attr =
    case uModel {
      UPinging  $\mapsto$  attrEmpty
      UWaiting  $\mapsto$  attr "disabled" "true"
    } in
  html
  <html>
  <body>
    <button {attr} onClick = { $\lambda()$ .Click}>
      Send Ping!
    </button>
  </body>
</html>

handleClick(model)  $\triangleq$ 
  case model {
    Pinging( $c$ )  $\mapsto$ 
      let cmd =
        cmdSpawn ( let  $c = \text{send}$  (Ping,  $c$ ) in
                    let ( $pong, c$ ) = receive  $c$  in
                      Ponged( $c$ ) in
          (Waiting, cmd)
      Waiting  $\mapsto$  (Waiting, cmdEmpty)
  }

update : (Message  $\times$  Model)  $\rightarrow$ 
  (Model  $\times$  Cmd(Message))
update  $\triangleq$   $\lambda(msg, model).$ 
  case msg {
    Click  $\mapsto$ 
      handleClick(model)
    Ponged( $c$ )  $\mapsto$ 
      handlePonged(model,  $c$ )
  }

extract : Model  $\rightarrow$  (Model  $\times$  UModel)
extract  $\triangleq$   $\lambda model.$ 
  case model {
    Pinging( $c$ )  $\mapsto$  (Pinging( $c$ ), UPinging)
    Waiting  $\mapsto$  (Waiting, UWaiting)
  }

handlePonged(model,  $c$ )  $\triangleq$ 
  case model {
    Pinging( $c'$ )  $\mapsto$ 
      cancel  $c'$ ;
      (Pinging( $c$ ), cmdEmpty)
    Waiting  $\mapsto$ 
      (Pinging( $c$ ), cmdEmpty)
  }

```

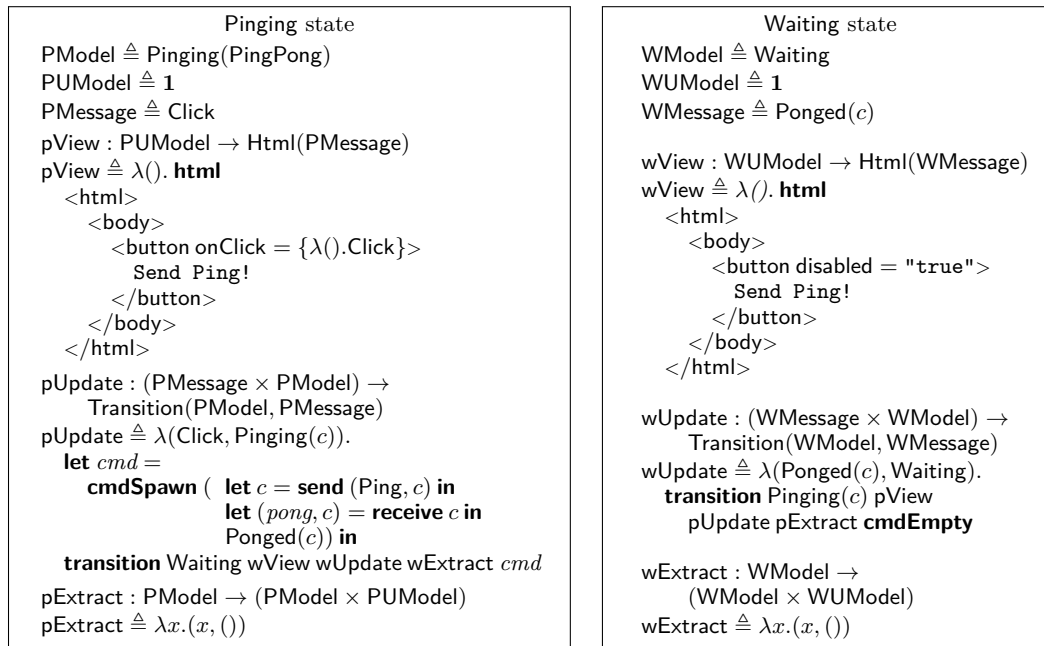
■ Figure 7 PingPong application.

Session types. Session types S range over output $!A.S$, input $?A.S$, the completed session End, recursive session types $\mu t.S$, and (possibly dualised) recursive type variables t . We take an equi-recursive treatment of recursive session types, identifying a recursive session type with its unfolding. We omit types and constructs for branching and selection as they can be encoded [28, 13]. The **send** constant sends a value of type A over an endpoint of type $!A.S$ and returns the continuation of the session, S . The **close** constant closes a completed session endpoint. The **receive** constant takes an endpoint of type $?A.S$ and receives a pair of a value of type A and endpoint of type S . The **cancel** constant allows an endpoint to be discarded safely [35, 20].

Session types $S ::= !A.S \mid ?A.S \mid \mu t.S \mid t \mid \bar{t} \mid \text{End}$

send : $(A \times !A.S) \rightarrow S$ **receive** : $?A.S \rightarrow (A \times S)$ **close** : $\text{End} \rightarrow \mathbf{1}$ **cancel** : $S \rightarrow \mathbf{1}$

Figure 7 shows the PingPong client written in λ_{MVU} . We can encode the PingPong protocol as a session type, $\mu t. !\text{Ping}. ?\text{Pong}. t$. The Model type encodes the two states of the application: Pinging(c) is the state where the "Send Ping!" button is enabled and the user can send a Ping message along session channel c , whereas Waiting is the state where the button is disabled and awaiting a Pong message from the other party. The UModel type is the unrestricted model type which does not include the session channel. Again, the Message type encodes the UI messages in the application: the Click UI message is produced when the button is pressed, whereas the Ponged(PingPong) UI message is produced when a Pong session message has been received. Note that the Ponged UI message now contains a session channel of type PingPong as a parameter.



■ **Figure 8** PingPong application using model transitions.

The view function takes an unrestricted model and displays a button, which is disabled in the **Waiting** state but enabled in the **Pinging** state. The **extract** function pairs the linear model with the associated unrestricted model.

The **update** function case-splits on the message. The **handleClick** function handles the **Click** message, and case-splits on the model. If the model is in the **Pinging**(c) state, then the function creates a command to spawn a process which will send a **Ping** message along c , receive a **Pong** message along c , and generate a **Ponged** UI message when the **Pong** message is received. The function finally updates the model to the **Waiting** state. If the model is in the **Waiting** state – which should not occur, since the button is disabled – then the model remains the same and no command is created.

The **handlePonged** function handles a **Ponged**(c) message. Again, we must case split on the model. If the model is in the **Waiting** state, then we can change to the **Pinging** state, given endpoint c . However, if the model is in the **Pinging**(c') state and a **Ponged** message is received – which should not occur, since according to the session type, there is no way for the peer to send a **Pong** message while we are waiting to send a **Ping** – we now have *two* linear resources. We choose to discard c' using **cancel**, and change the model to **Pinging**(c'), but this is an arbitrary choice to satisfy a code path that must exist, but should never be used.

3.3 Model transitions

Our proposal is still not quite satisfactory: as we saw with the **PingPong** example, we need to include cases in the **update** function which cannot arise. We highlight these in red. This is even more pronounced when dealing with linear resources, such as needing to handle a **Ponged** message when waiting to send a **Ping**.

The problem is that we are encoding the **Model** type using a sum type, whereas in fact we require *multiple* model types, and a way to *transition* between them.

Kinds	$\kappa ::= \mathbf{L} \mid \mathbf{U}$
Types	$A, B, C ::= \mathbf{1} \mid A \rightarrow^\kappa B \mid A \times B \mid A + B \mid \mathbf{String} \mid \mathbf{Int} \mid S$ $\mid \mathbf{Html}(A) \mid \mathbf{Attr}(A) \mid \mathbf{Cmd}(A) \mid \mathbf{Transition}(A, B)$
Session types	$S ::= !A.S \mid ?A.S \mid \mu t.S \mid t \mid \bar{t} \mid \mathbf{End}$
Terms	$L, M, N ::= x \mid \lambda x.M \mid M N \mid K M \mid () \mid s \mid n$ $\mid (M, N) \mid \mathbf{let} (x, y) = M \mathbf{in} N$ $\mid \mathbf{inl} x \mid \mathbf{inr} x \mid \mathbf{case} L \{ \mathbf{inl} x \mapsto M; \mathbf{inr} y \mapsto N \}$ $\mid \mathbf{htmlTag} \ t \ M \ N \mid \mathbf{htmlText} \ M$ $\mid \mathbf{attr} \ ak \ M \mid \mathbf{attrEmpty}$ $\mid \mathbf{cmdSpawn} \ M \mid \mathbf{cmdEmpty} \mid M \star N$ $\mid \mathbf{transition} \ M_m \ M_v \ M_u \ M_e \ M_c \mid \mathbf{noTransition} \ M_m \ M_c$ $\mid \mathbf{raise} \mid \mathbf{try} \ L \ \mathbf{as} \ x \ \mathbf{in} \ M \ \mathbf{otherwise} \ N$
Constants	$K ::= \mathbf{send} \mid \mathbf{receive} \mid \mathbf{new} \mid \mathbf{cancel} \mid \mathbf{close}$

■ **Figure 9** Syntax of extended calculus.

Example. Figure 8 shows how we can modify PingPong to use multiple model types. The left-hand side of the figure shows the Pinging state: the model type consists of the singleton variant tag `Pinging(PingPong)` containing an endpoint of type `PingPong`, the unrestricted model is the unit type, and the only message that the `Pinging` state can receive is `Click`. The `pView` function is similar to before, and the `pExtract` function returns a pair of the current state and the unit value. The `pUpdate` function is more interesting. Given the current state and a `Click` message, the function constructs a command which will send the `Ping` session message, receive the `Pong` session message, and then generate a `Ponged(c)` UI message containing the session channel. The function *transitions* into the `Waiting` state using the `transition` primitive, which allows the developer to specify new model, view, update, extract functions, and a command to evaluate. The functions for the `Waiting` state follow a similar pattern. Session types rule out the communication errors besetting the example in Figure 6, and model transitions eliminate the redundant code paths arising due to illegal states.

3.4 λ_{MVU} with Commands, Linearity, and Transitions

Commands, linearity, and transitions are the three key ingredients needed to extend MVU to support models which include session-typed channels. In this section, we introduce a calculus which combines all three extensions, and prove that the extended calculus is sound.

3.4.1 Syntax and Typing

Figure 9 shows the syntax of λ_{MVU} extended with commands, linearity, and transitions.

Types and kinds. To support linearity, types are assigned *kinds*, ranged over by κ . Types can either be *linear* (\mathbf{L}) or *unrestricted* (\mathbf{U}). A value of linear type must be used precisely once, whereas a value of unrestricted type can be used many times.

We modify function types to include a kind annotation: linear functions may close over linear variables and so must be used once. To support commands, we introduce type `Cmd(A)` which is the type of a command which produces messages of type `A`. To support transitions, we introduce type `Transition(A, B)` which is parameterised by the *current* model type `A` and message type `B`. Finally, we extend types to include session types `S` as described in §3.2.

Terms. Term `cmdSpawn M` is a command which can spawn term `M` as a thread, and is monoidally composable using \star and `cmdEmpty`.

Context splitting		$\boxed{\Gamma = \Gamma_1 + \Gamma_2}$
$\frac{}{\cdot = \cdot + \cdot}$	$\frac{A :: \mathbf{U}}{\Gamma, x : A = (\Gamma_1, x : A) + (\Gamma_2, x : A)}$	$\frac{}{\Gamma_1 + \Gamma_2, x : A = (\Gamma_1, x : A) + \Gamma_2}$
Modified typing rules for terms		$\boxed{\Gamma \vdash M : A}$
$\frac{\text{T-VAR}}{\Gamma :: \mathbf{U}} \frac{}{\Gamma, x : A \vdash x : A}$	$\frac{\text{T-ABS}}{\Gamma, x : A \vdash M : B} \frac{}{\Gamma \vdash \lambda x. M : A \rightarrow^\kappa B}$	$\frac{\text{T-APPK}}{\Sigma(K) = A \rightarrow^U B} \frac{}{\Gamma \vdash K M : B}$
$\frac{\text{T-CMD}}{\Gamma \vdash M : A} \frac{}{\Gamma \vdash \mathbf{cmdSpawn} M : \mathbf{Cmd}(A)}$	$\frac{\text{T-CMDEMPTY}}{\Gamma :: \mathbf{U}} \frac{}{\Gamma \vdash \mathbf{cmdEmpty} : \mathbf{Cmd}(A)}$	$\frac{\text{T-CMDAPPEND}}{\Gamma_1 \vdash M : \mathbf{Cmd}(A)} \frac{}{\Gamma_1 + \Gamma_2 \vdash M \star N : \mathbf{Cmd}(A)}$
$\frac{\text{T-TRANSITION}}{\Gamma_1 \vdash M_m : A} \frac{}{\Gamma_2 \vdash M_v : A \rightarrow^U \mathbf{Html}(B)} \frac{}{\Gamma_3 \vdash M_u : (B \times A) \rightarrow^U \mathbf{Transition}(A, B)} \frac{}{\Gamma_4 \vdash M_e : A \rightarrow^U (A \times C)} \frac{}{\Gamma_5 \vdash M_c : \mathbf{Cmd}(A)} \frac{}{C :: \mathbf{U}}$		
$\Gamma_1 + \dots + \Gamma_5 \vdash \mathbf{transition} M_m M_v M_u M_e M_c : \mathbf{Transition}(A', B')$		
$\frac{\text{T-EVTATTR}}{\Gamma \vdash M : \mathbf{ty}(h) \rightarrow^U A} \frac{}{\Gamma \vdash \mathbf{attr} h M : \mathbf{Attr}(A)}$	$\frac{\text{T-NOTRANSITION}}{\Gamma_1 \vdash M : A} \frac{}{\Gamma_2 \vdash N : \mathbf{Cmd}(B)} \frac{}{\Gamma_1 + \Gamma_2 \vdash \mathbf{noTransition} M N : \mathbf{Transition}(A, B)}$	
$\frac{\text{T-TRY}}{\Gamma_1 \vdash L : A} \frac{}{\Gamma_2, x : A \vdash M : B} \frac{}{\Gamma_2 \vdash N : B} \frac{}{\Gamma_1 + \Gamma_2 \vdash \mathbf{try} L \mathbf{as} x \mathbf{in} M \mathbf{otherwise} N : B}$	$\frac{\text{T-RAISE}}{\Gamma :: \mathbf{U}} \frac{}{\Gamma \vdash \mathbf{raise} : A}$	(other rules modified to split contexts)
Typing of constants		$\boxed{\Sigma(c)}$
$\Sigma(\mathbf{send}) = (A \times !A.S) \rightarrow^U S$	$\Sigma(\mathbf{receive}) = ?A.S \rightarrow^U (A \times S)$	$\Sigma(\mathbf{new}) = \mathbf{1} \rightarrow^U (S \times \bar{S})$
$\Sigma(\mathbf{cancel}) = S \rightarrow^U \mathbf{1}$	$\Sigma(\mathbf{close}) = \mathbf{End} \rightarrow^U \mathbf{1}$	
Duality		$\boxed{\bar{S}}$
$!A.\bar{S} = ?A.S$		$?A.\bar{S} = !A.S$
$\overline{\mu t.S} = \mu t.\bar{S}\{\bar{t}/t\}$		$\bar{\bar{t}} = t$
$\bar{\mathbf{End}} = \mathbf{End}$		

■ **Figure 10** Term typing for extended calculus.

There are two terms for transitions: the **noTransition** $M_m M_c$ term denotes that no transition is to occur, and that the model should be updated to M_m and command M_c should be evaluated; and **transition** $M_m M_v M_u M_e M_c$ denotes that a transition should occur, with new model M_m , view function M_v , update function M_u , extraction function M_e , and command M_c to be run once the transition has taken place.

To support session typing, we introduce session typing constants, ranged over by K , as described in §3.2. We also introduce an application form for constants, $K M$.

Finally, as discussed in §3.2, it is useful to be able to explicitly discard (or *cancel*) a session channel. In particular, cancellation is crucial in order to handle the interplay between linearity and transitions, as all unprocessed messages (which may contain linear resources) must be safely discarded when a transition occurs.

Following Mostrous and Vasconcelos [35] and Exceptional GV (EGV) by Fowler et al. [20], if a thread tries to receive from an endpoint whose peer has been cancelled, an exception is raised (**raise**). Exceptions can be handled using the **try** L **as** x **in** M **otherwise** N construct, which tries to evaluate term L , and binds the result to x in M if the term evaluates to a value, and evaluates N if the term raises an exception.

Kinding and subkinding. The kinding relation $A :: \kappa$ assigns kind κ to type A ; our formulation is inspired by that of Padovani [38]. Base types and HTML and attribute types are unrestricted. The kind of a function type is determined by its kind annotation. Session types are linear. The kinds of product, sum, command and transition types are determined by the kinds of their type parameters. The reflexive *subkinding* rule $U \leq L$ combined with the kinding subsumption rule states that if a value can be used many times, then it can also be treated as only being used once. We write $\Gamma :: \kappa$ if $A :: \kappa$ for each $x : A \in \Gamma$.

► **Definition 5** (Kinding and subkinding). *We define the subkinding relation as the reflexive relation on kinds \leq such that $U \leq L$. We define the kinding relation $A :: \kappa$ as the largest relation between types and kinds such that:*

- $A :: \kappa'$ if $A :: \kappa$ and $\kappa \leq \kappa'$
- $S :: L$
- $A :: U$ if $A \in \{\mathbf{1}, \text{String}, \text{Int}, \text{Html}(B), \text{Attr}(B)\}$
- $A \rightarrow^\kappa B :: \kappa$
- $\text{Cmd}(A) :: \kappa$ if $A :: \kappa$
- $C :: \kappa$ if $C \in \{A \times B, A + B, \text{Transition}(A, B)\}$ and both $A :: \kappa$ and $B :: \kappa$

Term typing. Figure 10 shows the typing rules for the extended calculus. The splitting relation $\Gamma = \Gamma_1 + \Gamma_2$ [8] splits a typing context Γ into two subcontexts which may share only unrestricted variables. We support linearity by changing T-VAR to only type a variable in an unrestricted context, and by using the context splitting judgement when typing subterms. The adaptation of the remaining rules to use context splitting is standard, so we omit them.

The constant application rule T-APPK types term $K M$ and makes use of the type schema function $\Sigma(K)$ to ensure that the argument M is of the correct type. Rule T-CMDSPAWN assigns term **cmdSpawn** M type $\text{Cmd}(A)$ if term M has type A , and rules T-CMDEMPTY and T-CMDAPPEND allow commands to be composed monoidally.

Rule T-TRANSITION types a **transition** term. The typing rule ensures that the types of the new model, and view, update and extract functions are compatible. Note that the type parameters of the $\text{Transition}(A', B')$ need not match the types of the new model and functions. Rule T-NOTRANSITION assigns term **noTransition** $M N$ type $\text{Transition}(A, B)$ if new model M has type A , and N is a command of type $\text{Cmd}(B)$. Note that in this way, the **noTransition** $M N$ term replaces the standard result of the **update** function.

Rule T-TRY types an exception handler: the continuations share a typing environment, but the success continuation is augmented with the a variable of the type of the possibly-failing continuation. Finally, **raise** can have any type as is it does not return (T-RAISE).

The type and kinding system ensures that the kind of type A determines the kind of the typing environment needed to type a term of type A .

► **Lemma 6.** *If $\Gamma \vdash M : A$ and $A :: \kappa$, then $\Gamma :: \kappa$.*

Duality. The duality relation for session types is standard: output types are dual to input types; we use a self-dual **End** type; and we use the formulation of the duality of recursive session types advocated by Lindley and Morris [32].

3.4.2 Operational Semantics

Runtime syntax. Figure 11 shows the runtime syntax for the combined calculus. We introduce *runtime names* c, d which identify session channel endpoints.

Runtime syntax	
Runtime names	c, d
Values	$U, V, W ::= \dots \mid c \mid \mathbf{cmdSpawn} M \mid \mathbf{noTransition} V W$ $\mid \mathbf{transition} V_m V_v V_u V_e V_c$
Active thread	$T ::= \mathbf{idle} V_m \mid \mathbf{updating} M \mid \mathbf{extracting}[V_c] M$ $\mid \mathbf{extractingT}[F, V_c] M \mid \mathbf{rendering}[V_m, V_c] M$ $\mid \mathbf{transitioning}[V_m, F, V_c] M$
Versions	ι
Processes	$P, Q ::= \mathbf{run} M \mid \langle T \mid F \rangle_\iota \mid ((M))_\iota \mid P \parallel Q$ $\mid (\nu cd)P \mid [M] \mid \cancel{c} \mid \mathbf{halt}$
Function state	$F ::= (V_v, V_u, V_e)$
Configurations	$\mathcal{C} ::= P \S D$
Process contexts	$\mathcal{P} ::= [] \mid \mathcal{P} \parallel P \mid (\nu cd)\mathcal{P}$
Evaluation contexts	$E ::= \dots \mid K E \mid \mathbf{noTransition} E M \mid \mathbf{noTransition} V E$ $\mid \mathbf{transition} E M_v M_u M_e M_c \mid \dots \mid \mathbf{transition} V_m V_v V_u V_e E$ $\mid \mathbf{try} E \mathbf{as} x \mathbf{in} M \mathbf{otherwise} N$
Pure contexts	$E_P ::=$ (as E , but without exception handling frames)
Active thread contexts	$\mathcal{T}_A ::= \mathbf{updating} E \mid \mathbf{rendering}[V_m, V_c] E \mid \mathbf{extracting}[V_c] E$ $\mid \mathbf{extractingT}[V_c, F'] E \mid \mathbf{transitioning}[V_m, F', V_c] E$
Pure active thread contexts	$\mathcal{T}_P ::=$ (as \mathcal{T}_A , but for pure contexts)
Thread contexts	$\mathcal{T} ::= \mathbf{run} E \mid \langle \mathcal{T}_A \mid F \rangle_\iota \mid ((E))_\iota \mid [E]$
Active thread state machine	
<pre> graph LR idle([idle]) -- "(Model transition)" --> updating([updating]) updating -- "(No model transition)" --> extracting([extracting]) updating -- "(Model transition)" --> extractingT([extractingT]) extracting --> rendering([rendering]) extractingT --> transitioning([transitioning]) </pre>	

■ **Figure 11** Runtime syntax for extended calculus.

The biggest departure is that we require a richer structure on active threads, which form a state machine based on whether a model transition occurs. The **idle** state is as before, and the **updating** state evaluates the update function. If there is no model transition, then the thread moves to the **extracting** state to extract the unrestricted model, and the **rendering** state to render the new HTML. If there is a model transition, then the thread moves to the **extractingT** state followed by the **transitioning** state to calculate the new HTML to be displayed after the transition. Each state records values which are required in later states: for example, the **rendering** $[V_m, V_c] M$ state records the new model V_m and the command to be executed upon updating the page V_c .

We introduce four new types of process. To model client-server communication, we introduce server processes $[M]$ to model a process M running on the server; the thread to spawn is given as an argument to **run**. As an example, we could write a Ponger server process for the PingPong example, which immediately responds with a Pong message:

<pre> let (c, s) = new () in (Ping(c), pView, pUpdate, pExtract, cmdEmpty, ponger(s)) </pre>	<pre> ponger(s) \triangleq λ(). (rec f(s) . let (Ping, s) = receive s in let s = send (Pong, s) in f s) s </pre>
---	---

A name restriction $(\nu cd)P$ binds runtime names c and d in process P , following the double-binder formulation due to Vasconcelos [43]. A *zapper thread* \cancel{c} denotes an endpoint c that has been cancelled and cannot be used in future communications; we write $\cancel{c}V$ to mean $\cancel{c}_1 \parallel \dots \parallel \cancel{c}_n$ for $c_i \in \text{fn}(V)$, where $\text{fn}(V)$ enumerates the free runtime names in a value V , and extend this sugar to evaluation contexts. The **halt** process denotes that the event loop process has terminated due to an unhandled exception.

<p>Additional term reduction rule $M \longrightarrow_M N$</p> <p>E-TRY $\text{try } V \text{ as } x \text{ in } M \text{ otherwise } N \longrightarrow_M M\{V/x\}$</p> <p>Equivalence of processes</p> <p>$(\nu cd)(\nu c' d')P \equiv (\nu c' d')(\nu cd)P$ $P \parallel ((\nu cd)Q) \equiv (\nu cd)(P \parallel Q)$ if $c, d \notin \text{fn}(P)$ $(\nu cd)P \equiv (\nu dc)P$</p> <p>$P_1 \parallel P_2 \equiv P_2 \parallel P_1$ $P_1 \parallel (P_2 \parallel P_3) \equiv (P_1 \parallel P_2) \parallel P_3$ $(\nu cd)(\zeta c \parallel \zeta d) \parallel P \equiv P$ $[\] \parallel P \equiv P$</p> <p>Reduction of processes $P \longrightarrow P'$</p>	<p>Additional meta-level definitions</p> <p>$\text{procs}(\text{cmdEmpty}) = \epsilon$ $\text{procs}(\text{cmdSpawn } M) = M$ $\text{procs}(V \star W) = \text{procs}(V) \cdot \text{procs}(W)$</p> <p style="text-align: right;">$P \equiv P'$</p> <p style="text-align: center;">MVU reduction rules</p> <p>E-DISCARD $\langle T \mid F \rangle_\iota \parallel ((V))_{\iota'} \longrightarrow \langle T \mid F \rangle_\iota \parallel \zeta V$ if $\iota \neq \iota'$ E-DISCARDHALT $\text{halt} \parallel ((V))_\iota \longrightarrow \text{halt} \parallel \zeta V$ E-HANDLE $\langle \text{idle } V_m \mid (V_v, V_u, V_e) \rangle_\iota \parallel ((V))_\iota \longrightarrow \langle \text{updating } V_u (V, V_m) \mid (V_v, V_u, V_e) \rangle_\iota$ E-EXTRACT $\langle \text{updating (noTransition } V_m V_c) \mid F \rangle_\iota \longrightarrow \langle \text{extracting}[V_c] (V_e V_m) \mid F \rangle_\iota$ where $F = (V_v, V_u, V_e)$ E-EXTRACTT $\langle \text{updating (transition } V_m V_v V_u V_e V_c) \mid F \rangle_\iota \longrightarrow \langle \text{extractingT}[(V_v, V_u, V_e), V_c] (V_e V_m) \mid F \rangle_\iota$ E-RENDER $\langle \text{extracting}[V_c] (V_m, V_{um}) \mid F \rangle_\iota \longrightarrow \langle \text{rendering}[V_m, V_c] (V_v V_{um}) \mid F \rangle_\iota$ where $F = (V_v, V_u, V_e)$ E-RENDERT $\langle \text{extractingT}[F', V_c] (V_m, V_{um}) \mid F \rangle_\iota \longrightarrow \langle \text{transitioning}[V_m, F', V_c] (V_v V_{um}) \mid F \rangle_\iota$ where $F' = (V_v, V_u, V_e)$</p> <p style="text-align: center;">Session reduction rules</p> <p>E-NEW $\mathcal{T}[\text{new}()] \longrightarrow (\nu cd)(\mathcal{T}[(c, d)])$ where c, d fresh E-COMM $(\nu cd)(\mathcal{T}[\text{send } (V, c)] \parallel \mathcal{T}'[\text{receive } d]) \longrightarrow (\nu cd)(\mathcal{T}[c] \parallel \mathcal{T}'[(V, d)])$ E-CLOSE $(\nu cd)(\mathcal{T}[\text{close } c] \parallel \mathcal{T}'[\text{close } d]) \longrightarrow \mathcal{T}[\] \parallel \mathcal{T}'[\]$ E-CANCEL $\mathcal{T}[\text{cancel } c] \longrightarrow \mathcal{T}[\] \parallel \zeta c$ E-SENDZAP $(\nu cd)(\mathcal{T}[\text{send } (V, c)] \parallel \zeta d) \longrightarrow (\nu cd)(\mathcal{T}[\text{raise}] \parallel \zeta c \parallel \zeta V \parallel \zeta d)$ E-RECVZAP $(\nu cd)(\mathcal{T}[\text{receive } c] \parallel \zeta d) \longrightarrow (\nu cd)(\mathcal{T}[\text{raise}] \parallel \zeta c \parallel \zeta d)$ E-CLOSEZAP $(\nu cd)(\mathcal{T}[\text{close } c] \parallel \zeta d) \longrightarrow (\nu cd)(\mathcal{T}[\text{raise}] \parallel \zeta c \parallel \zeta d)$</p> <p style="text-align: center;">Exception reduction rules</p> <p>E-RAISEH $\mathcal{T}[\text{try } E_P[\text{raise}] \text{ as } x \text{ in } M \text{ otherwise } N] \longrightarrow \mathcal{T}[N] \parallel \zeta E_P$ E-RAISEURUN $\text{run } (E_P[\text{raise}]) \longrightarrow \text{halt} \parallel \zeta E_P$ E-RAISEUMAIN $\langle \mathcal{T}_P[\text{raise}] \mid F \rangle_\iota \longrightarrow \text{halt} \parallel \zeta \mathcal{T}_P$ E-RAISEUTHREAD $((E_P[\text{raise}]))_\iota \longrightarrow \zeta E_P$ E-RAISEUSERVER $[E_P[\text{raise}]] \longrightarrow \zeta E_P$</p> <p style="text-align: center;">Administrative reduction rules</p> <p>E-LIFTT $\mathcal{T}[M] \longrightarrow \mathcal{T}[N]$ if $M \longrightarrow_M N$ E-NU $(\nu ab)P \longrightarrow (\nu ab)P'$ if $P \longrightarrow P'$ E-PAR $P_1 \parallel P_2 \longrightarrow P'_1 \parallel P_2$ if $P_1 \longrightarrow P'_1$</p>
--	---

■ **Figure 12** Reduction rules for extended calculus (1).

We extend evaluation contexts in the standard way, and introduce a class of *pure contexts* E_P , which are evaluation contexts which do not contain any exception handling frames.

Versions. *Versions* ι ensure that threads spawned in a previous state do not deliver incompatible messages. We annotate event loop processes and event handler threads with versions: given an event loop $\langle T \mid F \rangle_\iota$, a thread $((M))_{\iota'}$ where $\iota \neq \iota'$ can be of arbitrary type as it will be discarded. We write $\text{version}(P) = \iota$ if P contains a subprocess $\langle T \mid F \rangle_\iota$.

Reduction. Figures 12 and 13 show the extended process equivalence and reduction rules. Rule E-TRY handles evaluation of the success continuation of an exception handler, and the `procs` meta-definition returns a sequence of processes to be spawned by a command. Process equivalence is extended to allow commutativity of name restrictions, reordering of names in a binder, and scope extrusion. The final “garbage collection” equivalences $(\nu cd)(\zeta c \parallel \zeta d) \parallel P \equiv P$ and $[\] \parallel P \equiv P$ allow us to discard a channel where both endpoints have been cancelled, and a completed server thread, respectively.

Figure 12 details the extended MVU process reduction rules.

Reduction of configurations	$\mathcal{C} \longrightarrow \mathcal{C}'$
<p>E-RUN $\mathcal{P}[\mathbf{run}(V_m, V_v, V_u, V_e, V_c, \lambda().M)] \S D \longrightarrow \langle \mathbf{extracting}[V_c](V_e V_m) \mid (V_v, V_u, V_e) \rangle_0 \parallel [M] \S D$</p> <p>E-UPDATE $\mathcal{P}[\langle \mathbf{rendering}[V'_m, V_c] U \mid F \rangle_\iota] \S D \longrightarrow \mathcal{P}[\langle \mathbf{idle} V'_m \mid F \rangle_\iota \parallel ((M_1))_\iota \parallel \dots \parallel ((M_n))_\iota] \S D'$ where $\mathbf{diff}(U, D) = D'$ and $\mathbf{procs}(V_c) = \vec{M}$</p> <p>E-TRANSITION $\mathcal{P}[\langle \mathbf{transitioning}[V_m, F', V_c] U \mid F \rangle_\iota] \S D \longrightarrow \mathcal{P}[\langle \mathbf{idle} V_m \mid F' \rangle_{\iota'} \parallel ((M_1))_{\iota'} \parallel \dots \parallel ((M_n))_{\iota'}] \S D'$ where $\iota' = \iota + 1$, $\mathbf{diff}(U, D) = D'$ and $\mathbf{procs}(V_c) = \vec{M}$</p> <p>E-EVT $P \S \mathcal{D}[\mathbf{domTag}(\mathbf{ev}(W) \cdot \vec{e}) \mathbf{t} U D] \longrightarrow P \parallel ((V_1 W))_\iota \parallel \dots \parallel ((V_n W))_\iota \S \mathcal{D}[\mathbf{domTag}(\vec{e}) \mathbf{t} U D]$ where $\mathbf{handlers}(\mathbf{ev}, U) = \vec{V}$ and $\mathbf{version}(P) = \iota$</p>	
(E-INTERACT, E-STRUCT, E-LIFTP unchanged)	Cancellation of pure active thread contexts $\not\leq \mathcal{T}_P$
$\not\leq \mathbf{updating} E_P = \not\leq E_P$	$\not\leq \mathbf{rendering}[V_m, V_c] E_P = \not\leq V_m \parallel \not\leq V_c \parallel \not\leq E_P$
$\not\leq \mathbf{extracting} \mathbf{T}[V_c, F] E_P = \not\leq V_c \parallel \not\leq E_P$	$\not\leq \mathbf{transitioning}[V_m, F, V_c] E_P = \not\leq V_m \parallel \not\leq V_c \parallel \not\leq E_P$

■ **Figure 13** Reduction rules for extended calculus (2).

MVU reduction. MVU reduction rules are specific to MVU. Central to safely integrating linearity and transitions are rules E-DISCARD, E-DISCARDHALT, and E-HANDLE. Rule E-HANDLE is modified so that the event loop process only handles a message if the message has the same version. If the versions do not match, then E-DISCARD safely discards any channel endpoints in the discarded message by generating zipper threads. Rules E-EXTRACT, E-EXTRACTT, E-RENDER, and E-RENDERT handle the state machine transitions described in Figure 11 and are used to calculate the new model and HTML.

Session reduction. Session reduction rules encode session-typed communication and are mostly standard: E-NEW generates a name restriction and returns two fresh endpoints; E-COMM handles synchronous communication; and E-CLOSE discards the endpoints of a completed session. The remaining session communication rules handle session cancellation, and are a synchronous variant of Exceptional GV described by Fowler et al. [20]. Rule E-CANCEL discards an endpoint. Rules E-SENDZAP, E-RECVZAP, and E-CLOSEZAP raise an exception if a thread tries to communicate along an endpoint whose peer is cancelled, ensuring resources are discarded safely.

Exception reduction. Rule E-RAISEH describes exception handling: as **raise** occurs in a pure context, the exception is handled by the innermost handler; the rule evaluates the failure continuation and discards all linear resources in the aborted context. Rules E-RAISEURUN and E-RAISEU MAIN apply to unhandled exceptions in a main thread, generating the **halt** configuration and cancelling any linear resources in the aborted context. Rules E-RAISEU THREAD and E-RAISEU SERVER apply to unhandled exceptions in event loop thread and server threads respectively, by cancelling any channels in the aborted continuation.

Configuration reduction. Figure 13 shows the modified configuration reduction rules. We modify E-RUN to take into account the new arguments, and spawn the given server thread. We modify E-UPDATE to spawn threads described by the returned command; E-TRANSITION is similar but changes the function state and increments the version. We modify E-EVT to tag each spawned event handler thread with the version of the event handler process.

Typing rules for names, events, and function state				$\Gamma \vdash M : A$	$\vdash e$	$\Psi \vdash F : \text{State}(A, B, C)$
T-NAME $\frac{\Gamma :: \mathbf{U}}{\Gamma, c : S \vdash c : S}$	TE-EVT $\frac{\vdash V : \text{ty}(\text{ev})}{\text{ty}(\text{ev}) :: \mathbf{U}}$	TF-STATE $\frac{\Psi_1 \vdash V_v : A \rightarrow^{\mathbf{U}} \text{Html}(B) \quad \Psi_2 \vdash V_u : (B \times A) \rightarrow^{\mathbf{U}} \text{Transition}(A, B)}{\Psi_3 \vdash V_e : A \rightarrow^{\mathbf{U}} (A \times C) \quad C :: \mathbf{U}}$		$\Psi_1, \Psi_2, \Psi_3 \vdash (V_m, V_v, V_u) : \text{State}(A, B, C)$		
Typing rules for active threads				$\Psi \vdash T : \text{EvtLoop}(A, B, C)$		
TT-IDLE $\frac{\Psi \vdash V_m : A}{\Psi \vdash \text{idle } V_m : \text{EvtLoop}(A, B, C)}$				TT-UPDATING $\frac{\Psi \vdash M : \text{Transition}(A, B)}{\Psi \vdash \text{updating } M : \text{EvtLoop}(A, B, C)}$		
TT-RENDERING $\frac{\Psi_1 \vdash V_m : A \quad \Psi_2 \vdash V_c : \text{Cmd}(B) \quad \Psi_3 \vdash M : \text{Html}(B)}{\Psi_1, \Psi_2, \Psi_3 \vdash \text{rendering}[V_m, V_c] M : \text{EvtLoop}(A, B, C)}$				TT-EXTRACTING $\frac{\Psi_1 \vdash V_c : \text{Cmd}(B) \quad \Psi_2 \vdash M : (A \times C)}{\Psi_1, \Psi_2 \vdash \text{extracting}[V_c] M : \text{EvtLoop}(A, B, C)}$		
TT-EXTRACTINGT $\frac{\Psi_1 \vdash F : \text{State}(A, B, C) \quad \Psi_2 \vdash V_c : \text{Cmd}(B) \quad \Psi_3 \vdash M : (A \times C)}{\Psi_1, \Psi_2, \Psi_3 \vdash \text{extractingT}[F, V_c] M : \text{EvtLoop}(A', B', C')}$						
TT-TRANSITIONING $\frac{\Psi_1 \vdash V_m : A \quad \Psi_2 \vdash F : \text{State}(A, B, C) \quad \Psi_3 \vdash V_c : \text{Cmd}(B) \quad \Psi_4 \vdash M : \text{Html}(B)}{\Psi_1, \dots, \Psi_4 \vdash \text{transitioning}[V_m, F, V_c] M : \text{EvtLoop}(A', B', C')}$						
Typing rules for processes				$\Psi \vdash_i^\phi P : A$		
TP-RUN $\frac{\Psi \vdash M : (A \times (A \rightarrow^{\mathbf{U}} \text{Html}(B)) \times ((B \times A) \rightarrow^{\mathbf{U}} \text{Transition}(A, B)) \times (A \rightarrow^{\mathbf{U}} (A \times C)) \times \text{Cmd}(B) \times (\mathbf{1} \rightarrow^{\mathbf{L}} \mathbf{1}))}{\Psi \vdash_i^\bullet \text{run } M : B}$						
TP-EVENTLOOP $\frac{\Psi_1 \vdash T : \text{EvtLoop}(A, B, C) \quad \Psi_2 \vdash F : \text{State}(A, B, C)}{\Psi_1, \Psi_2 \vdash_i^\bullet \langle T F \rangle_i : B}$	TP-THREAD $\frac{\Psi \vdash M : A}{\Psi \vdash_i^\circ ((M))_i : A}$	TP-OLDTHREAD $\frac{\Psi \vdash M : B \quad i \neq i'}{\Psi \vdash_i^\circ ((M))_{i'} : A}$	TP-SERVERTHREAD $\frac{\Psi \vdash M : \mathbf{1}}{\Psi \vdash_i^\circ [M] : A}$			
TP-PAR $\frac{\Psi_1 \vdash_i^{\phi_1} P_1 : A \quad \Psi_2 \vdash_i^{\phi_2} P_2 : A}{\Psi_1, \Psi_2 \vdash_i^{\phi_1 + \phi_2} P_1 \parallel P_2 : A}$	TP-ZAP $\frac{c : S \vdash_i^\circ \not\leq c : A}{\cdot \vdash_i^\bullet \text{halt} : A}$	TP-HALT $\frac{\cdot \vdash_i^\bullet \text{halt} : A}{\cdot \vdash_i^\bullet \text{halt} : A}$	TP-NU $\frac{\Psi, c : S, d : \bar{S} \vdash_i^\phi P : A}{\Psi \vdash_i^\phi (\nu cd)P : A}$			

■ **Figure 14** Runtime typing for extended calculus.

3.4.3 Metatheory

Runtime typing. Figure 14 shows the runtime typing rules for the extended calculus. Rule T-NAME types channel endpoints, and rule TE-EVT mandates that event payload types are unrestricted. The rules for active threads ensure that the types of the terms being evaluated correspond to the state in the state machine (for example, that the **updating** state returns a term of type $\text{Transition}(A, B)$), and that any recorded values have the correct types.

Let Ψ range over environments containing only runtime names: $\Psi ::= \cdot \mid \Psi, c : S$. We write Ψ_1, Ψ_2 for the disjoint union of environments Ψ_1 and Ψ_2 .

We modify the shape of the process typing judgement to $\Psi \vdash_i^\phi P : A$, which can be read “under typing environment Ψ and thread flag ϕ , process P has type A and version i ”. We modify rule TP-EVENTLOOP to include the extraction function, and mandate that the unrestricted model type C has kind \mathbf{U} . We modify rule T-THREAD to state that type of an event handler thread $((M))_i$ has type A if term M has type A and the version matches that of the event handler process. Rule TP-OLDTHREAD allows a thread to have a mismatching

type to the event handler process if the versions are incompatible. Finally, TP-ZAP and TP-HALT type zapper threads and the **halt** thread, and TP-NU types a name restriction $(\nu cd)P$ by adding c and d with dual session types into the typing environment.

Properties. The extended calculus satisfies preservation.

► **Theorem 7** (Preservation). *If $\vdash C$ and $C \longrightarrow C'$, then $\vdash C'$.*

Although session types rule out deadlock within a single session, without imposing a tree-like structure on processes [45, 31] (which is too inflexible for our purposes) or using techniques such as channel priorities [37, 39, 29], it is not possible to rule out deadlocks when considering multiple sessions. Since communication over multiple sessions can introduce deadlocks, we begin by proving an error-freedom property, similar to that of Gay and Vasconcelos [21]. An *error process* involves a communication mismatch.

► **Definition 8** (Error process). *A process P is an error process if it contains one of the following processes as a subprocess:*

1. $(\nu cd)(\mathcal{T}[\mathbf{send}(V, c)] \parallel \mathcal{T}'[\mathbf{send}(W, d)])$
2. $(\nu cd)(\mathcal{T}[\mathbf{send}(V, c)] \parallel \mathcal{T}'[\mathbf{close} d])$
3. $(\nu cd)(\mathcal{T}[\mathbf{receive} c] \parallel \mathcal{T}'[\mathbf{receive} d])$
4. $(\nu cd)(\mathcal{T}[\mathbf{receive} c] \parallel \mathcal{T}'[\mathbf{close} d])$

Configuration typing ensures error-freedom.

► **Theorem 9** (Error-freedom). *If $\Psi \vdash_{\phi} P : A$, then P is not an error process.*

Error-freedom shows that session typing ensures the absence of communication mismatches. What remains is to show that, apart from the possibility of deadlock, the additional features do not interfere with the progress property enjoyed by λ_{MVU} . We begin by classifying the notion of a *blocked* thread, which is a thread blocked on a communication action.

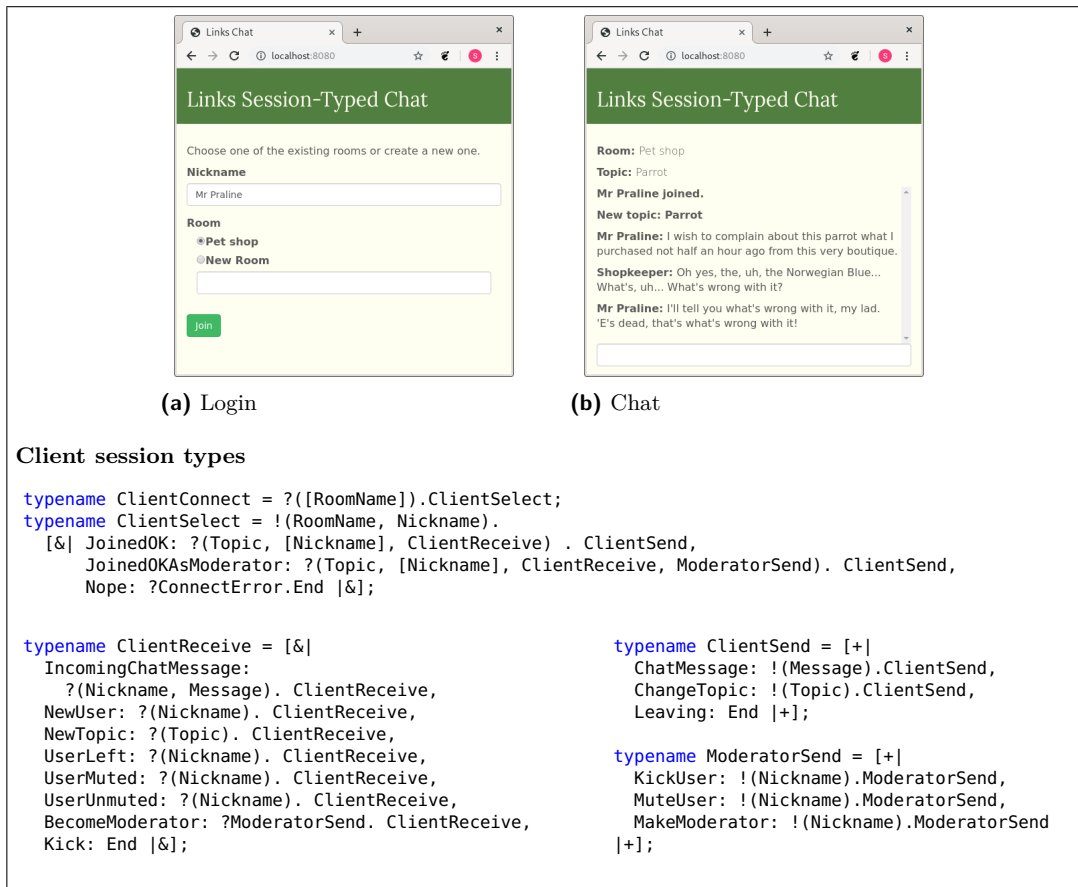
► **Definition 10** (Blocked thread). *We say a thread $\mathcal{T}[M]$ is blocked if either $M = \mathbf{send}(V, W)$, $M = \mathbf{receive} V$, or $M = \mathbf{close} V$.*

Let us refer to **halt**, $\langle T \mid F \rangle_{\iota}$, and **run** M as *main threads*, and $((M))_{\iota}$, $[M]$, and ζc as *auxiliary threads*. Each well-typed configuration has precisely one main thread.

We can now classify the notion of progress enjoyed by the extended calculus. Either the configuration can reduce; is waiting for an event; has halted due to an unhandled exception; or is deadlocked. Again, let $\longrightarrow_{\text{E}}$ be the \longrightarrow relation without E-INTERACT.

► **Theorem 11** (Weak Event Progress). *Suppose $\vdash C$. Either there exists some C' such that $C \longrightarrow C'$, or there exists some C' such that $C \equiv C'$ and:*

1. D cannot be written $\mathcal{D}[\mathbf{domTag}(\vec{e}) \text{ t } V D]$ for a non-empty \vec{e} .
2. If the main thread of C' is **halt**, then all auxiliary threads are blocked or zapper threads.
3. If the main thread of C' is **run** M , then M is blocked, and all auxiliary threads are either blocked, values, or zapper threads.
4. If the main thread of C' is $\langle T \mid F \rangle_{\iota}$, then:
 - a. if $T = \mathbf{idle} V_m$, then each auxiliary thread is either blocked or a zapper thread; or
 - b. if $T = \mathcal{T}_A[L]$ then L is blocked, and each auxiliary thread is either blocked, a value, or a zapper thread.



■ **Figure 15** Chat server application.

4 Implementation and Example Application

We have implemented an MVU library for the Links tierless web programming language, which includes all extensions in the paper; Links already has a linear type system and distributed session types, so is an ideal fit.

We now describe a chat application, extending the application presented by Fowler et al. [20]. The application (Figure 15) has two main stages shown to the user: on the first, the user is presented with a list of rooms, and enters a username and selects a room. If a user with the given nickname is not already in the selected room, then the user joins the room, receiving the current topic, a list of other nicknames, and a channel used to receive messages from the server. The user can then send chat messages, change the topic, and leave the room. If the user is the first user in the room, then they join as a moderator and receive an additional channel which can be used to kick, mute, or promote other users to moderators. Users can receive incoming chat messages, and system messages detailing changes such as a new topic or a user joining the room.

We can encode these interaction patterns using session types. Links session type notation for offering a choice is $[&| \dots |&]$, and making a choice is $[+| \dots |+]$. Type `ClientConnect` describes the client receiving the room list. Type `ClientSelect` describes the client sending the room name and nickname, and receiving the response from the server: either joining as a regular user (`JoinedOK`); joining as a moderator (`JoinedOKAsModerator`); or an error (`Nope`). Types `ClientSend` and `ClientReceive` detail the messages that the client can send to, and receive from the server, respectively. Type `ModeratorSend` details privileged moderator actions.

Although the original version of Links [10] ran as a CGI script, modern Links applications run as a persistent webserver. Upon execution, the chat application creates an *access point* for sessions of type `ClientConnect`, which supports session establishment, and spawns an acceptor thread to accept incoming requests on the access point. Each chat room is represented as a process on the server. When an HTTP request is made, the response contains the MVU application and the access point ID which can be used to establish a session of type `ClientConnect`. After the initial HTTP response, further communication between the client and server happens over a `WebSocket` [16].

The application has three states: connection, chatting, and a “waiting” state shown while waiting for a response. For the purposes of the paper, we consider the connection state.

```

typename SelectedRoom =
  [| NewRoom | SelectedRoom: String |];
typename NotConnectedModel =
  (nickname: String, rooms: [RoomName],
   selectedRoom: SelectedRoom,
   newRoomText: RoomName, error: Maybe(Error));
typename NCMModel =
  (ClientSelect, NotConnectedModel);
typename NCMessage = [|
  UpdateNickname: Nickname
  | UpdateSelectedRoom: SelectedRoom
  | UpdateNewRoom: RoomName | SubmitJoinRoom |];

```

The `NotConnectedModel` is the unrestricted part of the model, and contains the current nickname (`nickname`), list of rooms (`rooms`), selected room (`selectedRoom`), value of the “new room” text box (`newRoomText`), and an optional error message to display (`error`). The model, `NCModel`, is a pair of a session endpoint of type `ClientSelect` and a `NotConnectedModel`. The UI messages are described by the `NCMessage` type: for example, the `UpdateNickname` message is generated by the `onInput` event of the nickname input box.

Upon receiving the `SubmitJoinRoom` UI message when the form is submitted, the application can send the nickname and selected room along the `ClientSelect` channel, all of which are contained in the model, without requiring ad-hoc messaging or imperative updates.

5 Related work

Flapjax [34] was the first web programming language to use *functional reactive programming* (FRP) [14] in the setting of web applications. Flapjax provides *behaviours*, which are variables whose contents change over time, and *event streams*, which are an infinite stream of discrete events which change a behaviour. `ScalaLoc` [46] is a multi-tier reactive programming framework written in Scala, where changes in reactive *signals* are propagated across tiers, rather than using explicit message passing. `Ur/Web` [9] and `WebSharper UI` [19] store data in mutable variables, and allow views of the data to be combined using monadic combinators.

Felleisen et al. [15] describe an earlier approach similar to MVU written in the `DrScheme` [17] system. Similar to the MVU update function, events such as key presses and mouse movements are handled using functions of type $(\text{Model} \times \text{Event}) \rightarrow \text{Model}$. The approach handles “environment” events rather than events dispatched by individual elements, and the approach is not formalised. Environment events can be handled using *subscriptions* in Elm, which can be added to λ_{MVU} (see the extended version of the paper [18]).

React [2] is a popular JavaScript UI framework. In React, a user defines data models and rendering functions, and similar to Elm, updates are propagated to the DOM by diffing. Differently to MVU, there is no notion of a message, and a page consists of multiple components rather than being derived from a single model. We expect some technical machinery from λ_{MVU} (e.g., event queues, DOM contexts, and diffing) could be reused when formalising React. `Redux` [5] is a state container for JavaScript applications: to modify the state, one dispatches an *action*, and a function takes the previous state and an action and produces a new state. In combination with React, the approach strongly resembles MVU.

Hop.js [41] is a multi-tier web framework written in JavaScript. Hop.js *services* allow remote function invocation, and the framework supports client-side message-passing concurrency using Web Workers [22], but there is no cross-tier message-passing concurrency.

Session types were introduced by Honda [23] and were first considered in a linear functional language by Gay and Vasconcelos [21]; Wadler [45] later introduced a session-typed functional language GV and a logically-grounded session-typed calculus CP (following Caires and Pfenning [7]), and translated GV into CP. Lindley and Morris [31] introduced an operational semantics for GV, and showed type- and semantics-preserving translations between GV and CP. GV inspires FST [33], which is the core calculus for Links' treatment of session typing.

Fowler et al. [20] extend GV with failure handling, and extend Links with cross-tier session-typed communication. They do not formally consider GUI development, and their approach to frontend web programming using session types (described in Section 1) leads to a disconnect between the state of the page and the application logic. We build upon their approach to session-typed web programming, while also allowing idiomatic GUI development.

King et al. [27] present a toolchain for writing web applications which respect multiparty session types [25]. Protocols are compiled to PureScript [42] using a parameterised monad [6] to guarantee linearity, and the authors integrate their encoding of session types with the Concur UI framework [26]. Each application may only have a single session connecting the client and server, whereas in our system there may be multiple; our approach supports first-class linearity and cross-tier typechecking; our approach is formalised; and our approach supports failure handling. Links does not yet support multiparty session types.

6 Conclusion

Session types allow conformance to protocols to be checked statically. The last few years have seen a flurry of activity in implementing session types in a multitude of programming languages, but linearity – a vital prerequisite for implementing session types safely – is difficult to reconcile with the asynchronous nature of graphical user interfaces. Consequently, the vast majority of implementations using session types are command line applications, and the few implementations which do integrate session types and GUIs do so in an ad-hoc manner.

In this paper, we have addressed this problem by extending the Model-View-Update architecture, pioneered by the Elm programming language. We have presented the first formal study of MVU by introducing a core calculus, λ_{MVU} . Leveraging our formal characterisation of MVU, we have introduced three extensions: commands, linearity, and model transitions, enabling us to present the first formal integration of session-typed communication with a GUI framework. Informed by our formal model, we have implemented our approach in Links. As future work, we will investigate how to encode allowed transitions as a behavioural type.

References

- 1 Elm: A delightful language for reliable webapps, 2019. Accessed on 2019-07-04. URL: <http://www.elm-lang.org>.
- 2 React – a JavaScript library for building user interfaces, 2019. Accessed on 2019-09-02. URL: <http://www.reactjs.org>.
- 3 WebSharper.Mvu, 2019. Accessed on 2019-07-04. URL: <https://github.com/dotnet-websharpervmvu>.
- 4 Flux, 2020. Accessed on 2020-01-08. URL: <https://facebook.github.io/flux/>.
- 5 Redux - a predictable state container for JavaScript apps, 2020. Accessed on 2020-01-08. URL: <https://redux.js.org/>.

- 6 Robert Atkey. Parameterised notions of computation. *J. Funct. Program.*, 19(3-4):335–376, 2009.
- 7 Luís Caires and Frank Pfenning. Session types as intuitionistic linear propositions. In *CONCUR*, volume 6269 of *Lecture Notes in Computer Science*, pages 222–236. Springer, 2010.
- 8 Iliano Cervesato and Frank Pfenning. A linear logical framework. In *LICS*, pages 264–275. IEEE Computer Society, 1996.
- 9 Adam Chlipala. Ur/Web: A simple model for programming the web. In *POPL*, pages 153–165. ACM, 2015.
- 10 Ezra Cooper, Sam Lindley, Philip Wadler, and Jeremy Yallop. Links: Web programming without tiers. In *FMCO*, volume 4709 of *Lecture Notes in Computer Science*, pages 266–296. Springer, 2006.
- 11 Evan Czaplicki. Farewell to FRP, 2016. Accessed on 2019-09-02. URL: <https://elm-lang.org/news/farewell-to-frp>.
- 12 Evan Czaplicki and Stephen Chong. Asynchronous functional reactive programming for GUIs. 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 411–422. ACM, 2013. doi:10.1145/2491956.2462161.
- 13 Ornela Dardha, Elena Giachino, and Davide Sangiorgi. Session types revisited. *Inf. Comput.*, 256:253–286, 2017.
- 14 Conal Elliott and Paul Hudak. Functional reactive animation. In *ICFP*, pages 263–273. ACM, 1997.
- 15 Matthias Felleisen, Robert Bruce Findler, Matthew Flatt, and Shriram Krishnamurthi. A functional I/O system or, fun for freshman kids. In *ICFP*, pages 47–58. ACM, 2009.
- 16 Ian Fette and Alexey Melnikov. The WebSocket protocol, 2011.
- 17 Robert Bruce Findler, John Clements, Cormac Flanagan, Matthew Flatt, Shriram Krishnamurthi, Paul Steckler, and Matthias Felleisen. DrScheme: a programming environment for Scheme. *J. Funct. Program.*, 12(2):159–182, 2002.
- 18 Simon Fowler. Model-View-Update-Communicate: Session types meet the Elm architecture (Extended version), 2019. arXiv:1910.11108.
- 19 Simon Fowler, Loïc Denuzière, and Adam Granicz. Reactive single-page applications with dynamic dataflow. In *PADL*, volume 9131 of *Lecture Notes in Computer Science*, pages 58–73. Springer, 2015.
- 20 Simon Fowler, Sam Lindley, J. Garrett Morris, and Sára Decova. Exceptional asynchronous session types: session types without tiers. *PACMPL*, 3(POPL):28:1–28:29, 2019. doi:10.1145/3290341.
- 21 Simon J. Gay and Vasco Thudichum Vasconcelos. Linear type theory for asynchronous session types. *J. Funct. Program.*, 20(1):19–50, 2010.
- 22 Ido Green. *Web Workers - Multithreaded Programs in JavaScript*. O’Reilly, 2012.
- 23 Kohei Honda. Types for dyadic interaction. In *CONCUR*, volume 715 of *Lecture Notes in Computer Science*, pages 509–523. Springer, 1993.
- 24 Kohei Honda, Vasco Thudichum Vasconcelos, and Makoto Kubo. Language primitives and type discipline for structured communication-based programming. In *ESOP*, volume 1381 of *Lecture Notes in Computer Science*, pages 122–138. Springer, 1998.
- 25 Kohei Honda, Nobuko Yoshida, and Marco Carbone. Multiparty asynchronous session types. *J. ACM*, 63(1):9:1–9:67, 2016.
- 26 Anupam Jain. Concur, 2019. Accessed on 2019-09-02. URL: <https://ajnsit.github.io/concur>.
- 27 Jonathan King, Nicholas Ng, and Nobuko Yoshida. Multiparty session type-safe web development with static linearity. In *PLACES@ETAPS*, volume 291 of *EPTCS*, pages 35–46, 2019.
- 28 Naoki Kobayashi. Type systems for concurrent programs. In *10th Anniversary Colloquium of UNU/IIST*, volume 2757 of *Lecture Notes in Computer Science*, pages 439–453. Springer, 2002.

- 29 Naoki Kobayashi. A new type system for deadlock-free processes. In *CONCUR*, volume 4137 of *Lecture Notes in Computer Science*, pages 233–247. Springer, 2006.
- 30 Glenn E. Krasner and Stephen T. Pope. A Cookbook for using the Model-view Controller user interface paradigm in Smalltalk-80. *J. Object Oriented Program.*, 1(3):26–49, August 1988. URL: <http://dl.acm.org/citation.cfm?id=50757.50759>.
- 31 Sam Lindley and J. Garrett Morris. A semantics for propositions as sessions. In *ESOP*, volume 9032 of *Lecture Notes in Computer Science*, pages 560–584. Springer, 2015.
- 32 Sam Lindley and J. Garrett Morris. Talking bananas: structural recursion for session types. In *ICFP*, pages 434–447. ACM, 2016.
- 33 Sam Lindley and J. Garrett Morris. Lightweight functional session types. *Behavioural Types: from Theory to Tools*. River Publishers, pages 265–286, 2017.
- 34 Leo A. Meyerovich, Arjun Guha, Jacob P. Baskin, Gregory H. Cooper, Michael Greenberg, Aleks Bromfield, and Shriram Krishnamurthi. Flapjax: a programming language for Ajax applications. In *OOPSLA*, pages 1–20. ACM, 2009.
- 35 Dimitris Mostrous and Vasco T. Vasconcelos. Affine sessions. *Logical Methods in Computer Science*, 14(4), 2018. doi:10.23638/LMCS-14(4:14)2018.
- 36 Joachim Niehren, Jan Schwinghammer, and Gert Smolka. A concurrent lambda calculus with futures. *Theor. Comput. Sci.*, 364(3):338–356, 2006.
- 37 Luca Padovani. Deadlock and lock freedom in the linear π -calculus. In *CSL-LICS*, pages 72:1–72:10. ACM, 2014.
- 38 Luca Padovani. Context-free session type inference. In *ESOP*, volume 10201 of *Lecture Notes in Computer Science*, pages 804–830. Springer, 2017.
- 39 Luca Padovani and Luca Novara. Types for deadlock-free higher-order programs. In *FORTE*, volume 9039 of *Lecture Notes in Computer Science*, pages 3–18. Springer, 2015.
- 40 Adam Pedley. Functional Model-View-Update Architecture for Flutter, 2019. Accessed on 2019-09-24. URL: <https://buildflutter.com/functional-model-view-update-architecture-for-flutter/>.
- 41 Manuel Serrano and Vincent Prunet. A glimpse of Hopjs. In *ICFP*, pages 180–192. ACM, 2016.
- 42 The PureScript Contributors. PureScript, 2019. Accessed on 2019-09-02. URL: <http://www.purescript.org/>.
- 43 Vasco T. Vasconcelos. Fundamentals of session types. *Inf. Comput.*, 217:52–70, 2012.
- 44 Philip Wadler. Linear types can change the world! In *Programming Concepts and Methods*, page 561. North-Holland, 1990.
- 45 Philip Wadler. Propositions as sessions. *J. Funct. Program.*, 24(2-3):384–418, 2014. doi:10.1017/S095679681400001X.
- 46 Pascal Weisenburger, Mirko Köhler, and Guido Salvaneschi. Distributed system development with ScalaLoc. *PACMPL*, 2(OOPSLA):129:1–129:30, 2018.

Static Analysis of Shape in TensorFlow Programs

Sifis Lagouvardos

University of Athens, Greece
sifis.lag@di.uoa.gr

Julian Dolby

IBM Research, Yorktown Heights, NY, USA
dolby@us.ibm.com

Neville Grech

University of Athens, Greece
me@nevillegrech.com

Anastasios Antoniadis

University of Athens, Greece
anantoni@di.uoa.gr

Yannis Smaragdakis

University of Athens, Greece
smaragd@di.uoa.gr

Abstract

Machine learning has been widely adopted in diverse science and engineering domains, aided by reusable libraries and quick development patterns. The TensorFlow library is probably the best-known representative of this trend and most users employ the Python API to its powerful back-end. TensorFlow programs are susceptible to several systematic errors, especially in the dynamic typing setting of Python. We present Pythia, a static analysis that tracks the shapes of tensors across Python library calls and warns of several possible mismatches. The key technical aspects are a close modeling of library semantics with respect to tensor shape, and an identification of violations and error-prone patterns. Pythia is powerful enough to statically detect (with 84.62% precision) 11 of the 14 shape-related TensorFlow bugs in the recent Zhang et al. empirical study – an independent slice of real-world bugs.

2012 ACM Subject Classification Theory of computation → Program analysis; Software and its engineering → Compilers; Software and its engineering → General programming languages

Keywords and phrases Python, TensorFlow, static analysis, Doop, Wala

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2020.15

Supplementary Material ECOOP 2020 Artifact Evaluation approved artifact available at <https://doi.org/10.4230/DARTS.6.2.6>.

Funding We gratefully acknowledge funding by the European Research Council, grant 790340 (PARSE), and by the Hellenic Foundation for Research and Innovation (project DEAN-BLOCK).

1 Introduction

Machine learning has seen widespread use in recent years, for an enormous variety of application domains, from vision to language processing to programming tasks [3, 23, 39] and well beyond, into mainstream science and engineering. The TensorFlow library [1], originally developed by the Google Brain Team, is the dominant open-source framework for modern machine learning applications. TensorFlow has received significant attention and impressive adoption, continually extending its dominance over other frameworks. Current statistics (as



© Sifis Lagouvardos, Julian Dolby, Neville Grech, Anastasios Antoniadis, and Yannis Smaragdakis;
licensed under Creative Commons License CC-BY

34th European Conference on Object-Oriented Programming (ECOOP 2020).

Editors: Robert Hirschfeld and Tobias Pape; Article No. 15; pp. 15:1–15:29



Leibniz International Proceedings in Informatics

Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany



15:2 Static Analysis of Shape in TensorFlow Programs

of Jan.08, 2020) show the TensorFlow GitHub repository with over 140K stars and 79.4K forks, with other popular open-source frameworks for machine learning lagging far behind (PyTorch [37] at 35.2K stars and 8.8K forks, Theano [2] at 9K stars and 2.5K forks).

As might be expected, TensorFlow programs are not free of defects (“bugs”). In high-level code, such as TensorFlow clients, bugs are commonly due to misunderstandings of the guarantees offered and obligations imposed by increasingly layered software. At the same time, such bugs have increasing real-world importance, as machine learning makes advances in widespread adoption. In a recent empirical survey, Zhang et al. [58] collect and classify a variety of TensorFlow program bugs from StackOverflow QA page and GitHub projects, by examining documentation, informal posts, commit and pull-request messages, and issue discussions. Many of these bugs are semantic in nature: they can only be ascertained by inspecting the outcome or the performance of the underlying computation. Others are bugs that may admit automatic detection: they signify API misuse, often (but not always) triggering assertions during execution.

TensorFlow, as many other popular machine learning frameworks, is mostly used from Python: a dynamic language that offers significant flexibility and ease of adoption. The dynamic nature of Python implies that there is no static tracking of types that can be used to ensure compatibility of values and operations. Furthermore, the static analysis tools available for Python are less advanced than those in statically-typed languages, focusing more on local code issues rather than whole-program properties. One reason for this has been a lack of underlying general analysis frameworks (analogous, e.g., to WALA [50], Soot [52], or Doop [9] in the Java world) that deploy whole-program technology and support Python. (For instance, we have failed to find a publicly available library for points-to analysis of Python programs.)

In this work, we focus on a class of TensorFlow bugs that relate to the *shape* of tensors, i.e., the number of their dimensions and the dimensions’ sizes. Checking that the shape of tensor arguments is compatible with the expectations of library operators is a key validation technique. Shape checking can prevent a large and important class of real-world TensorFlow programming errors, including the 14 shape-related bugs identified in StackOverflow questions by Zhang et al. [58].

Our approach tracks the shape of tensors using static analysis of the Python program and appropriate modeling of the TensorFlow API. In addition to the dynamism of the Python language, static analysis or type checking of TensorFlow code is also hindered by the inherent dynamism of the library itself. The design philosophy of the library (much in line with its common use from a dynamic language) is that of being very resilient to incomplete data. The API exhibits multiple instances of dynamic padding, reshaping, unknown dimensions, partially-known shapes (to be filled in dynamically), and more. Our analysis follows the flexibility of the library operators and attempts to closely model what is a permitted and expected behavior vs. what will produce a run-time error or is very likely a logical error and should induce a warning.

The work offers both application-level and technical-level contributions:

- We define Pythia, a state-of-the-art static analysis for the modeling of tensor shape through TensorFlow API calls. The analysis combines several elements: a relatively complete front-end translating Python source code into the IR of the WALA framework; a translation of the WALA IR into a relational representation for defining analyses using declarative Datalog rules; a whole-program context-sensitive value-flow and points-to analysis for Python; and a shape analysis of tensor values that carefully captures the flexibility of library operators.

- We provide the first concrete demonstration of the applicability of static analysis in the TensorFlow domain, by showing that our tool can find real bugs in real TensorFlow programs. We validate the effectiveness of the analysis by applying it to the 14 shape-related bug examples (and their fixed versions) in the [58] study. Pythia correctly finds 11 of these bugs with a precision of 84.62% and recall of 78.6%. (Importantly, of the missed bugs, all but one are undetectable with static information alone.)
- We present insights on the design of static shape checking for Python/TensorFlow programs. In particular, we argue that an effective such analysis is best classified as a static analysis and not a type checker, due to its desired features (extensional, non-modular behavior, context sensitivity).

2 Background

We next present background useful in later sections, on TensorFlow and Datalog program structure.

TensorFlow

TensorFlow is the most widely-adopted open-source machine learning library. The library performs computations using symbolic data-flow graphs. Operators form the vertices of the graph and tensors are flowing along the edges. TensorFlow invocation from Python code typically follows a two-stage pattern.¹ Initially the data-flow graph representing the computations is constructed. The entire graph is in place before dynamic data have been read. This graph or a number of its sub-graphs can then be executed multiple times with different input data.

During the construction phase of the graph the information about each tensor’s shape may vary. It may range from fully-known or *concrete*, to partially-known (where one or more dimensions is unknown, represented as `None`) to completely unknown. The static analysis we describe is based on retrieving as much shape information as possible from the program text, and propagating it through TensorFlow operators, which require careful modeling with respect to their shape transformations. Therefore, *the analysis is crucially based on common TensorFlow programming patterns*. These encourage encoding known shape information in the program text, while leaving unknown (dynamic) shape information undefined.

Datalog in Program Analysis

The Datalog language has been often used to declaratively specify static analysis algorithms [8, 18, 20, 25, 27, 30, 33, 36, 46, 53, 54, 57]. We use Datalog in our analysis, both in the high-level description and in its implementation, in order to seamlessly combine the results of several separate analyses (constant-flow, points-to, tensor-shape), with each one appealing to others.

A Datalog program is a set of logical inference rules, operating over initial facts and producing more inferences until fixpoint. A rule “ $C(z,x) \leftarrow A(x,y), B(y,z)$.” means that if $A(x,y)$ and $B(y,z)$ are both true, then $C(z,x)$ can be inferred. We shall use syntactic shorthands in the rules, such as multiple rule heads (“ $H1(\dots), H2(\dots) \leftarrow \dots$ ”), which are equivalent to repeating the rule for each head, and disjunction (operator “ $;$ ”) in the rule body, which is equivalent to replicating the body for each disjunct.

¹ This description, as well as all of our work and presentation, applies to TensorFlow v.1.X, the most widely deployed version of the framework. TensorFlow v.2 was released in late 2019 and includes a radical (and incompatible) reworking of the programming model. Both our core analysis and the engineering scaffolding need to be reworked to apply to TensorFlow v.2, which will likely give rise to related but not identical kinds of bugs. This is a potentially promising future work direction.

3 Illustration: TensorFlow Shape Tracking

The concept of a tensor’s shape is straightforward and mostly well-understood: every tensor has a list of dimensions, each with a size. Tensor operations are well-defined when the arguments’ dimensions match the operator’s expectations. We shall see in Section 4 a more complete mathematical modeling of tensor shapes, but a simple, well-known example is the 2-dimensional tensor (matrix) multiplication operator (TENSOR $i\ j$ represents a tensor of shape $i \times j$):

$$\text{mul2d} : \text{TENSOR } i\ j \rightarrow \text{TENSOR } j\ k \rightarrow \text{TENSOR } i\ k$$

The complexity of modeling tensor shape in practice is much greater, however. The issue is precisely the dynamism that the TensorFlow library (as well as the Python language) affords. Our analysis seeks to capture this flexibility while closely modeling shape transformations through the TensorFlow API. We next consider several examples that illustrate a) how placeholder tensors, reshaping operations, implicit padding, and subtle semantic differences affect shape reasoning; b) which behaviors cause crashes and which can be reasonably considered likely bugs, and should, therefore, also elicit a warning; c) what flavor an analysis should adopt to capture such bugs in realistic programs.

Example 1: Placeholders

A first example helps demonstrate “placeholder” tensors.

```
import tensorflow as tf
import numpy as np
data1 = np.random.normal(0, 0.1, [20, 50])
data2 = np.random.normal(0, 0.1, [50])
a = tf.placeholder("float", shape=[None, 50])
b = tf.placeholder("float")
y = tf.matmul(a,b)
with tf.Session() as sess:
    print(sess.run(y, feed_dict={a:data1,b:data2}))
```

A placeholder tensor is a tensor that will be fed data at runtime. At instantiation of a placeholder tensor, some dimensions (or the whole shape) can be set to `None`, as in tensors `a` and `b` in our example. Feeding data to a placeholder can be done using the `feed_dict` optional argument to `Session.run()`, `Tensor.eval()`, or `Operation.run()`. When one or more dimensions are set to `None`, the data fed to this tensor has to match the shape of the placeholder, meaning that the number of dimensions has to be the same and the sizes of all explicit-sized dimensions should be equal.

The most common pattern is to set a dimension that represents the “batch number” of the data to `None`, to support placeholder tensors with a variable batch size: the structure of each instance is known, but the total number of instances is a run-time variable. In the code snippets we will be showcasing throughout the paper, the arguments of TensorFlow and NumPy² operations that affect the output shape will be highlighted in red. Such is the case in our example, where the call to `np.random.normal()` results in `data1` pointing to a NumPy array object with shape `[20,50]`. Consequently, feeding `data1` to the placeholder `a` with shape `[None,50]` is successful. In our static analysis, this placeholder operation will produce two different modeled result values: `[None,50]` and `[20,50]`.

² NumPy is the dominant Python scientific computing package. Our modeling also covers parts of NumPy that are particularly relevant to TensorFlow operations.

Placeholder tensors with no initial shape can be fed data of any shape, as long as the types are compatible. To model these, we take advantage of the shape of the data fed to the placeholder whenever it is available. In our example, the `b` placeholder tensor gets the shape of `data2` which is `[50]`. The call to `tf.matmul()` will fail with an error due to the two argument tensors having different number of dimensions. Our static analysis will issue an error, since no combination of the modeled abstract values for tensors `a` and `b` yields a compatible pair.

Example 2: Reshaping

Not all tensorflow bugs will result in run-time exceptions/assertion failures, yet strong evidence may exist that the code contains an error. An example is below, also illustrating the `tf.reshape()` operator.

```
import tensorflow as tf
import numpy as np
a = tf.placeholder(tf.float32, [None,784])
data = np.random.normal(0, 0.1, [36, 784])
b = tf.reshape(a, [-1,24,24,1])
with tf.Session() as sess:
    print(sess.run(b,feed_dict={a:data}).shape)
```

The `tf.reshape()` function attempts to reshape a tensor, given as input the dimensions specified by its second (`shape`) argument. In order for it to succeed, the product of the elements of the shape list of the input tensor (p_{in}) and the product of the elements of the shape list of the output tensor (p_{out}) should be equal. A very common special case concerns argument shape lists with a single allowed `-1` dimension, as in the `reshape` call of the example. The size of that dimension is then computed so that the reshape operation succeeds, provided that the product of explicit (i.e., not `-1`) dimensions of the shape argument is a divisor of p_{in} .

In the above example, just as in the earlier Example 1, the placeholder tensor `a` has originally one `None` dimension, corresponding to the batch size. The tensor, with shape `[None,784]`, is fed data with shape `[36,784]`. Dynamically, this value is compatible with the `reshape` operation, with attempted shape `[-1,24,24,1]`: the resulting shape of tensor `b` is `[49,24,24,1]`, since $49 \times 24 \times 24 \times 1 = 36 \times 784$. However, there is already a strong hint that the reshaping should only affect the second dimension, with size 784 (i.e., that the programmer expects that 784 should be divisible by 24): the batch size is a volatile attribute of the current input and not an inherent part of the tensor structure, as the explicit `[None,784]` shape suggests.

Our analysis keeps both abstract values, `[None,784]` and `[36,784]`, for tensor `a` and, since one of them is incompatible with the `reshape` operation, it emits a warning. Generally, when the input tensor of a `reshape` has one `None` dimension, we compute the products of the elements of the two shape lists excluding `None` and `-1` and if they are not equal we report a warning.

Example 3: Padding in Broadcast Operations

The distinction between analysis-reported errors and warnings is more generally meaningful for operations that are probably valid, yet likely to have surprising semantics. The most common such case is the “broadcasting” semantics of NumPy arrays. We discuss the behavior in Section 5 but the example below illustrates briefly.

```

import tensorflow as tf
x = tf.constant([[1.0, 1.0], [1.0, 2.0],
                [1.0, 3.0]], dtype=tf.float64)
y_ = tf.constant([1.0, 2.0, 3.0], dtype=tf.float64)
w = tf.truncated_normal(shape=[2,1], stddev=0.1, dtype=tf.float64)
y = tf.matmul(x, w)
diff = y - y_
error = tf.reduce_mean(tf.square(diff))

```

In this example, tensor x has a shape of $[3,2]$ and tensor w has a shape of $[2,1]$. Their product, tensor y has a shape of $[3,1]$. Tensor $y_$ has a shape of $[3]$. The difference of y ($[3,1]$) and $y_$ ($[3]$) has a shape of $[3,3]$, which is highly surprising to many users! (Broadcasting semantics copy leading dimensions of the higher-rank argument³ and match the rest one-to-one, expanding any dimension with size 1 to the size of the matching dimension from the other argument.)

Pythia models and correctly propagates the effect of broadcasting on shape. However, it produces a warning when array broadcasting results in the expansion of the dimensions of a tensor. This can help prevent errors caused by mechanics that can easily confuse a user.

4 Basic Tensor Shape Modeling

Practical analysis applications that yield realistic benefits need to devote considerable modeling effort to support the idiosyncrasies of different environments – in our case, TensorFlow’s operations. Much of the complexity of this modeling is due to technicalities employed for usability, to the sheer number of operators, or to the way data values are introduced from the host language. There is, however, a core set of operations that are representative of many more and whose basic shape modeling can be cleanly expressed in closed-form mathematical formulas, much as the reader might expect. We discuss the “clean” modeling of such operators in this section, and postpone discussing the more operational aspects of our analysis until Section 5. Therefore, this section is purposely simplifying, in order to ensure that the core model is clear to the reader. For instance, we omit tensors of partially-known shape (with `None` dimensions), special (-1) dimensions in reshaping, modeling of broadcasting, and other such complexities.

Every tensor operation is modeled mainly in terms of the output shape in relation to the inputs supplied to its formal parameters, and of the data type of individual tensor elements. Complexity mostly arises out of the former, so our design is influenced by this consideration. Tensor operations broadly consist of (i) shape pass-through functions, e.g., `identity`; (ii) convolution and pooling functions; (iii) conversions and reshaping from tensors or tensor-like objects (e.g., NumPy arrays).

Shape types of tensors are modeled using the following vocabulary for tensor types and tensor shapes.

$$\begin{array}{ll}
 \tau, v \in \textit{TensorType} & ::= \text{ TENSOR } T \\
 T, U \in \textit{DimensionType} & ::= T \ i \\
 & \quad | \ \text{NIL} \\
 & \quad i, j \in \mathbb{N}
 \end{array}$$

³ The “rank” of a shape is its number of dimensions.

We typically omit `NIL` for conciseness. Hence an example of a two dimensional tensor shape type is `TENSOR i j` , i.e., a tensor of shape $i \times j$.

Note that the variables in the above syntax are meta-variables, used for conceptual modeling. In concrete instances, inside our analysis, all shapes and their dimensions are concrete (i.e., sequences of integers or single integers, respectively). Conceptually, however, the logic of the analysis does use such meta-variables, since it handles any concrete numbers found in the program text. For instance, we can model the understanding of the analysis regarding the core TensorFlow operator `MUL` as follows.

$$\text{MUL} : \text{ TENSOR } U \ i \ j \rightarrow \text{ TENSOR } U \ j \ k \rightarrow \text{ TENSOR } U \ i \ k$$

`MUL` generalizes standard two-dimensional tensor (matrix) multiplication, by adding arbitrary (but identical, in both arguments and in the result) leading dimensions.

Similarly, the core operator `IDENTITY` takes a tensor of any shape and returns a copy of it with the same shape.

$$\text{IDENTITY} : \text{ TENSOR } T \rightarrow \text{ TENSOR } T$$

`RESHAPE` is another core TensorFlow operator. It takes a tensor of any shape T and tries to return a tensor of another shape U , supplied as argument.

$$\text{RESHAPE} : \text{ TENSOR } T \rightarrow U \rightarrow \text{ TENSOR } U$$

The `RESHAPE` operation succeeds if the product of all elements in T is equal to the product of all elements in U :

$$\prod_i T_i = \prod_i U_i$$

To get a glimpse of more complex and versatile shape modeling, still easy to express in a closed-form formula, we can consider the `CONV2D` operator – a core operator for convolution. Convolution is often used to create complex neural networks that can extract intermediate features (typically from an image), as part of an intermediate layer. Convolution takes a 4d `input` tensor, where the middle 2 dimensions represent the data, a `filter` tensor, and a `strides` shape. Furthermore, there are two padding strategies for convolution: `same` and `valid`. Essentially, the former pads the tensor with off-boundary data so that the convolution filter is still applicable on the edges, while the latter avoids padding and applies only up to the point where the filter still retrieves data from the input tensor. If the padding strategy is `same` the shape type of `CONV2D` is defined as:

$$\text{CONV2D} : \text{ TENSOR } * \ i \ j \ * \rightarrow \text{ TENSOR } * \ k \ l \ * \rightarrow * \ s_1 \ s_2 \ * \rightarrow \text{ TENSOR } \left[\frac{i}{s_1} \right] \left[\frac{j}{s_2} \right]$$

(The stars denote any, ignored, integer values.)

Otherwise, if the padding strategy is `valid`, the convolution shape type is defined as:

$$\text{CONV2D} : \text{ TENSOR } * \ i \ j \ * \rightarrow \text{ TENSOR } * \ k \ l \ * \rightarrow * \ s_1 \ s_2 \ * \rightarrow \text{ TENSOR } \left[\frac{i-k+1}{s_1} \right] \left[\frac{j-l+1}{s_2} \right]$$

5 Analysis Structure

Our analysis emphasis is on shape modeling, which is the main element of this work. However, given the dearth of static analysis infrastructure for Python, our analysis had to develop several techniques and combine them in a coherent whole: a Python front-end (parser, IR

generator) that produces intermediate code using the WALA framework [50], a generator of relational tables for declarative program analysis in the Doop framework [9], a points-to, constant-flow and call-graph analysis for Python.

We start our presentation from these underlying analyses, and proceed with representative fragments of the declarative modeling of shape transformations through TensorFlow operators.

5.1 Substrate: WALA and Declarative Value-Flow Analysis

Pythia is expressed declaratively, as Datalog rules for both value-flow and tensor-shape reasoning. For Python support, we extended the parser and intermediate-representation generator of the Ariadne system [12], which produces WALA IR statements from Python source. The past WALA front-end for Python was largely a proof-of-concept implementation, therefore several elements needed to be added to tackle realistic programs, for example:

- correct handling of the global scope of Python programs
- complete modeling of collections
- complete modeling of list comprehensions
- modeling of list slicing
- modeling of parameter initial values
- handling of constant values.

The resulting intermediate representation using WALA data structures is used to output tables for relational processing by Datalog-based analyses. We integrate the input relations generation and subsequent analysis with the Doop framework [9], which already features a WALA front-end and a declarative analysis scaffolding. Doop is a framework for analysis of Java bytecode – to add Python support, we implement a whole-program, context-sensitive value-flow analysis on the Python IR.

The form of this analysis is largely conventional, expressed using a standard declarative approach (e.g., [49]) over an SSA intermediate language (for flow sensitivity on local variables). The analysis propagates constants and object values inter-procedurally, maintaining precision using call-site sensitivity [47, 48]. (In the default setting, a 1-call-site-sensitive analysis with a context-sensitive heap is used, after experimentation with options to balance performance and precision.) A call-graph is inferred based on the values of receiver objects at method calls. The analysis is complete for the static features of Python, but several dynamic features (e.g., decorators, non-trivial list comprehensions, `eval/input`, `getattr`) will interrupt the propagation of values.

5.2 Declarative Modeling of Shape Transformations

The main analysis logic is expressed as rules that appeal to the substrate analysis of value-flow throughout the Python program. In general, the declarative model of the analysis helps in having simple, independent rules, mutually recursive with other sub-analyses. We illustrate two sample sets of rules, next, capturing shape reasoning for broadcast operations and reshape operations. As hinted in earlier examples, much of the complexity is due to the close modeling of the flexibility afforded by TensorFlow operators.

5.2.1 Broadcast Reasoning

Example 3 in Section 3 discussed array/tensor broadcasting. Array/tensor broadcasting is a mechanism to allow element-wise operations between arrays of different shapes. Under some restrictions, the smaller array is “broadcast” across the larger one, provided that

their dimensions match. Broadcasting operations can either be overloaded arithmetic binary operations, or calls to tensorflow functions (for example `tf.add()`, `tf.multiply()` or `tf.equal()`).

Consider an example to illustrate different cases:

```
import tensorflow as tf
op1 = tf.ones(shape=[4,3,1])
op2 = tf.ones(shape=[3,2])
res = tf.add(op1, op2)
```

The shape of the resulting tensor will be `[4,3,2]`: leading dimensions are “inherited” from the higher-rank tensor, and dimensions equal to 1 for either argument are expanded to the size of the corresponding dimension for the other argument.

Our analysis logic first creates a new value for a broadcasting operation, encoding (as a 3-tuple) the instruction and tensor argument values. We choose to have the operand with the higher rank as the first operand. The difference of the ranks is computed as the operation’s *offset*. In our example, the operation’s offset will be 1.

```
BROADCASTINGOP(bcastOp, offset) ←
  INVOCATION(insn, fun),
  BROADCASTINGFUNCTION(fun),
  ACTUALPARAMVALUE(insn, "x", tensor1),
  ACTUALPARAMVALUE(insn, "y", tensor2),
  TENSORRANK(tensor1, rank1),
  TENSORRANK(tensor2, rank2),
  rank1 >= rank2, offset = rank1 - rank2,
  bcastOp = [ insn, tensor1, tensor2 ].
```

The rule checks the preconditions of broadcasting and packages all relevant information for further processing. The relations in the rule body are produced by syntactic processing of the program text or by the global value-flow/points-to analysis: `INVOCATION(insn, fun)` recognizes a call to *fun* in instruction *insn* (such invocation resolution requires global value-flow reasoning); `BROADCASTINGFUNCTION` matches TensorFlow API functions that support broadcasting, such as `add` in our example; `ACTUALPARAMVALUE(insn, var, val)` computes the (abstract) value for the actual parameter (*var*, identified by name) of a call at instruction *insn*; `TENSORRANK` retrieves the rank of a tensor value (i.e., number of dimensions in its shape).

In words, the rule says that if two tensor values, *tensor1* and *tensor2*, are used as arguments of a broadcasting call, the call instruction, the tensor values, and the offset to be used to match the tensors’ dimensions are packaged in predicate `BROADCASTINGOP`.

Armed with the above, we can encode the different cases of shape propagation through broadcasting operators. The `RESULTSHAPEDIMENSION` predicate represents the contents of the shape list for each dimension of a broadcasting operation’s result.

For dimensions only in the higher-rank argument (i.e., below “*offset*”) the result inherits the size of the higher-rank argument’s dimension:

```
RESULTSHAPEDIMENSION(bcastOp, index, dim) ←
  BROADCASTINGOP(bcastOp, offset),
  bcastOp = [ _, tensor1, _ ],
  index < offset,
  TENSORSHAPE(tensor1, tensorShape1),
  SHAPEDIMENSION(tensorShape1, index, dim).
```

15:10 Static Analysis of Shape in TensorFlow Programs

Predicate `TENSORSHAPE` holds the shape of a tensor value, while `SHAPEDIMENSION(shape, i, dim)` holds the size, *dim*, of the *i*-th dimension of the shape value.

In our example, the above rule will produce the dimension with size 4 in the result.

In order to attempt a match of dimension sizes that are expected to match during a broadcasting operation, we introduce a convenience predicate, `ARGUMENTSSHAPEDIMENSIONS` that recalls both sizes at positions at least equal to *offset*:

```
ARGUMENTSSHAPEDIMENSIONS(bcastOp, index, dim1, dim2) ←  
  BROADCASTINGOP(bcastOp, offset),  
  bcastOp = [ _, tensor1, tensor2 ],  
  index ≥ offset,  
  TENSORSHAPE(tensor1, tensorShape1),  
  SHAPEDIMENSION(tensorShape1, index, dim1),  
  TENSORSHAPE(tensor2, tensorShape2),  
  SHAPEDIMENSION(tensorShape2, index - offset, dim2).
```

For fully matching argument dimensions, the common size becomes the size of the output dimension, as well (as in the second dimension of the output in our example):

```
RESULTSHAPEDIMENSION(bcastOp, index, dim) ←  
  ARGUMENTSSHAPEDIMENSIONS(bcastOp, index, dim, dim).
```

There are two more cases and they elicit a warning or an error report: The first computes an output shape of the resulting tensor for dimensions that are not equal but can match due to broadcasting. For two different dimensions to match in this way, at least one would need to be 1. This rule produces the result of the third dimension of our earlier example. In this case we also produce a warning, detecting the use of broadcasting mechanics that could confuse the user.

```
WARNING(bcastOp),  
RESULTSHAPEDIMENSION(bcastOp, index, dim) ←  
  ARGUMENTSSHAPEDIMENSIONS(bcastOp, index, dim1, dim2),  
  dim1 ≠ dim2,  
  ((dim1 = 1, dim = dim2) ; (dim2 = 1, dim = dim1)).
```

The final rule produces an error in the case of dimensions that cannot match, i.e., they are not equal and neither of them is 1.

```
ERROR(bcastOp) ←  
  ARGUMENTSSHAPEDIMENSIONS(bcastOp, index, dim1, dim2),  
  dim1 ≠ dim2, dim1 ≠ 1, dim2 ≠ 1.
```

5.2.2 Reshape Reasoning

Example 2 in Section 3 discussed the complexity of modeling the `reshape` operator in a realistic setting, unlike the core, closed-form modeling of Section 4. Tensors of partially-known shape (with `None` dimensions) and special reshape dimensions (of size -1) need to be accounted for in an analysis that aims to be useful for real-world bug detection. The Datalog rules we present next reflect these considerations.

First, as in broadcast operations, we identify calls to `reshape` and encode each instance of the operation as a new value, consisting of a 3-tuple of the instruction, tensor, and shape arguments:

```

RESHAPEOPERATION(rshpOp) ←
  INVOCATION(insn, "reshape"),
  ACTUALPARAMVALUE(insn, "tensor", tensorVal),
  ACTUALPARAMVALUE(insn, "shape", dimListVal),
  rshpOp = [insn, tensorVal, dimListVal].

```

We also store the products of dimension sizes for the tensor and the shape argument (with the result of the multiplication over all indexes computed separately – from rules not shown – into DIMENSIONSPRODUCT):

```

PRODUCTSOFSHAPES(rshpOp, tensorProd, dimListProd) ←
  RESHAPEOPERATION(rshpOp),
  rshpOp = [_, tensorVal, dimListVal],
  TENSORSHAPE(tensorVal, tensorShapeVal),
  DIMENSIONSPRODUCT(tensorShapeVal, tensorProd),
  DIMENSIONSPRODUCT(dimListVal, dimListProd).

```

We can then distinguish different cases of reshaping. A concrete-dimension tensor (i.e., with no `None` dimensions) reshaped into a concrete shape (i.e., with no `-1` dimensions) will succeed if the products of dimension sizes are equal and will produce an error otherwise. Predicate `RESHAPECONCRETETOCONCRETE` is used to cache intermediate results for use in the two later rules. “!” designates negation in a rule and in this case is used to establish the two shape concreteness conditions.

```

RESHAPECONCRETETOCONCRETE(rshpOp, tensorProd, dimListProd) ←
  PRODUCTSOFSHAPES(rshpOp, tensorProd, dimListProd),
  rshpOp = [_, tensorVal, dimListVal],
  TENSORSHAPE(tensorVal, tensorShapeVal),
  !SHAPEDIMENSION(tensorShapeVal, _, "None"),
  !SHAPEDIMENSION(dimListVal, _, -1).

TENSOROPERATIONPRODUCEOUTPUT(rshpOp) ←
  RESHAPECONCRETETOCONCRETE(rshpOp, tensorProd, tensorProd).

ERROR(rshpOp) ←
  RESHAPECONCRETETOCONCRETE(rshpOp, tensorProd, dimListProd),
  tensorProd != dimListProd.

```

Accordingly, we can handle the case of a concrete tensor resized to a special shape – i.e., one that has a `-1` dimension. (Other rules, omitted, enforce that there can be at most one `-1` dimension.) We first collect the products of sizes into a convenience predicate that also enforces the rest of the preconditions:

```

RESHAPECONCRETETOSPECIAL(rshpOp, tensorProd, dimListProd) ←
  PRODUCTSOFSHAPES(rshpOp, tensorProd, dimListProd),
  rshpOp = [_, tensorVal, dimListVal],
  TENSORSHAPE(tensorVal, tensorShapeVal),
  !SHAPEDIMENSION(tensorShapeVal, _, "None"),
  SHAPEDIMENSION(dimListVal, _, -1).

```

Subsequently, we distinguish the case of a correct reshaping, when the two dimension-size-products are divisible, from the error case, when they are not:

15:12 Static Analysis of Shape in TensorFlow Programs

```
TENSOROPERATIONPRODUCESOUTPUT(rshpOp) ←  
  RESHAPECONCRETETOSPECIAL(rshpOp, tensorProd, dimListProd),  
  quot = tensorProd/dimListProd,  
  tensorProd = quot * dimListProd.
```

```
ERROR(rshpOp) ←  
  RESHAPECONCRETETOSPECIAL(rshpOp, tensorProd, dimListProd),  
  tensorProd % dimListProd != 0.
```

In a largely similar fashion, we need to handle the case of a tensor of partially-known shape (with a `None` dimension) being reshaped to a shape with a special (-1) dimension. We first compute the products of concrete dimension sizes:

```
RESHAPEPARTIALTOSPECIAL(rshpOp, tensorProd, dimListProd) ←  
  PRODUCTSOFSHAPES(rshpOp, tensorProd, dimListProd),  
  rshpOp = [ _, tensorVal, dimListVal ],  
  TENSORSHAPE(tensorVal, tensorShapeVal),  
  SHAPEDIMENSION(tensorShapeVal, _, "None"),  
  SHAPEDIMENSION(dimListVal, _, -1).
```

Then, we distinguish the case of a correct reshaping, when both products match vs. one that elicits a warning, when they do not. This is the case of the earlier Example 2, commonly corresponding to a programming error. In the example, our analysis correctly infers the product of the input tensor to be 784 and that of the given shape argument to be 576, successfully reporting the appropriate warning. Remember that, for both shapes, their products are the products of the explicit dimensions (i.e. no `None` and -1 dimensions).

```
TENSOROPERATIONPRODUCESOUTPUT(rshpOp) ←  
  RESHAPEPARTIALTOSPECIAL(rshpOp, tensorProd, tensorProd).
```

```
WARNING(rshpOp),  
TENSOROPERATIONPRODUCESOUTPUT(rshpOp) ←  
  RESHAPEPARTIALTOSPECIAL(rshpOp, tensorProd, dimListProd),  
  tensorProd != dimListProd.
```

The final case is that of a tensor of partially-known shape reshaped into a concrete shape list, with no special dimensions. We again enforce the preconditions and cache the products of dimension sizes in a convenience predicate:

```
RESHAPEPARTIALTOCONCRETE(rshpOp, tensorProd, dimListProd) ←  
  PRODUCTSOFSHAPES(rshpOp, tensorProd, dimListProd),  
  rshpOp = [ _, tensorVal, dimListVal ],  
  TENSORSHAPE(tensorVal, tensorShapeVal),  
  SHAPEDIMENSION(tensorShapeVal, _, "None"),  
  !SHAPEDIMENSION(dimListVal, _, -1).
```

Subsequently, we distinguish the case of a correct reshaping from that of an error:

```
TENSOROPERATIONPRODUCESOUTPUT(rshpOp) ←  
  RESHAPEPARTIALTOCONCRETE(rshpOp, tensorProd, dimListProd),  
  dimListProd % tensorProd = 0.
```

```
ERROR(rshpOp) ←  
  RESHAPEPARTIALTOCONCRETE(rshpOp, tensorProd, dimListProd),  
  dimListProd % tensorProd != 0.
```

5.3 Tensor Value Representation

The presentation of the analysis so far has ignored the exact nature of abstract values that arise for tensors. In our previous rules, abstract tensor values are treated as black-box representations that have at least a type and a shape (a list of constants), which the rest of the analysis looks up. The exact abstraction of values, however, has significant implications on precision and performance (even up to non-termination, as Section 5.4 discusses). Our full analysis features two different configurations for tensor value creation: one very coarse and one highly precise.

The coarse value abstraction, which we term *simple-tensor-precision*, creates a single value for each function invocation instruction that resolves to a modeled tensor operation. The problem of this approach is that it can exacerbate the – sometimes unavoidable – imprecision of a static analysis. The snippet below provides a minimal example.

```
import tensorflow as tf
if(CONDITION):
    reshapeTo = [-1,28,28,1]
else:
    reshapeTo = [-1,14,14,4]
a = tf.placeholder(tf.int32, [None, 784])
a = tf.reshape(a, reshapeTo)
```

After the `reshape` operation, variable `a` points to one tensor value with one corresponding shape value. However, having a single value (for all dynamic instances of the operation) entails having a single shape list. This shape list has two possible values for each dimension except the 0th, combining the possible dimension values of the two run-time shape lists to a total of 8 possible shapes.

In contrast, the precise value abstraction of Pythia, which we call *full-tensor-precision*, represents each tensor value as the concatenation of all the values of arguments of the operation that creates it and the function invocation instruction that resolves to the operation.

For the above example, after the `reshape` operation, variable `a` points to two tensor values – one for each possible value of `reshapeTo` – but each with definite shape: the full shapes (`[-1,28,28,1]` vs. `[-1,14,14,4]`) of the `reshapeTo` arguments are kept in the two abstract values representing the operation’s results.

As discussed next, it is easy to switch between the two abstractions to implement interesting hybrid algorithms that give a balance of precision and scalability.

5.4 Analysis Termination

Termination is an interesting question regarding our analysis. There are new tensor shapes produced for several TensorFlow operators, e.g., by replacing `-1` dimension sizes with positive integers in the `reshape` operation. Also, even though the analysis deals with concrete dimensions, it remains a static analysis: a single variable can have many potential abstract values. These do not necessarily reflect dynamic values – they could arise due to control-flow or data-flow imprecision, i.e., because of an over-approximation. Therefore a program with no threat of non-termination can still possibly give rise to a non-terminating static analysis.

To ensure termination of our analysis, we first need to bound the new shapes that can be created. Doing so immediately establishes that our analysis will always terminate when running under the *simple-tensor-precision* value abstraction. We then show how a run of our analysis in the *simple-tensor-precision* configuration can ensure the termination of our analysis with the *full-tensor-precision* value abstraction for the same input program.

5.4.1 Finite shapes

The first challenge for establishing the finiteness of shapes is to show that the integer constants that arise (for each shape dimension independently) are finite.

Conveniently, tensor operations by themselves (without arithmetic in the Python program) cannot create an infinite number of dimension sizes. The dimension sizes for a new shape are either sizes of an existing tensor shape’s dimension (as in the case of tensor multiplication), or smaller dimension sizes (as in the case of convolution or reshaping operations, which take quotients of existing shape sizes).

Still, the above observation does not help bound the overall dimension sizes due to Python arithmetic. The finiteness property is actually one that the analysis needs to artificially enforce, since we propagate integer constants (corresponding to tensor dimension sizes) through arbitrary arithmetic operations. For instance, a tensor operation such as:

```
n_input = train_X.shape[1]
```

means that the number of inputs (which will later be used as the dimension size of a tensor) comes from the dimensions of another tensor. With arithmetic over the `n_input` variable and a looping construct, the potential dimension sizes become infinite. Therefore, we artificially bound the “complexity” (i.e., number of intermediate arithmetic operations) of the computed integer constants. (For instance, in our implementation, this bound is a generous 50.)

We additionally need to bound the maximum number of dimensions of a tensor, since operations such as `tf.expand_dims` can increase the number of dimensions.

5.4.2 Termination for Different Value Abstractions and Maximizing Precision

Based on the finiteness of integer dimensions in shapes, the analysis will always terminate for the *simple-tensor-precision* value abstraction: there is a finite number of values, each (by definition) has a single shape list, the shape list has a finite number of dimensions, and each dimension can have a finite set of values for its potential size.

The case for the *full-tensor-precision* value abstraction is more complicated. In principle, this abstraction is not finite: new tensor values can keep arising, even if they have the same shape. As an example, consider a `transpose(x, [0, 2, 1])` operation – permuting the dimensions of tensor argument `x` according to the list given as the second argument – with the output of the operation feeding back to the input tensor (due to a loop or recursion). In the *full-tensor-precision* abstraction, with output values of tensor operations being represented by the concatenation of all their input values, this would result in the creation of a `transpose(x, [0, 2, 1])` value feeding back to the argument of the operation, resulting in a `transpose(transpose(x, [0, 2, 1]), [0, 2, 1])` value, and so on.

Therefore, we employ the full-tensor-precision abstraction in our analysis only over tensor operations with no cyclic dependencies (on themselves). Concretely, Pythia first runs under a *simple-tensor-precision* abstraction, while also propagating values to detect circularity in the inference. For each tensor operation we compute the set of tensor values that flow to it and the corresponding operations that created them. In this way, we can detect the existence of cyclic dependencies. The simple-tensor-precision abstraction is less (i.e., at most as) precise than full-tensor-precision, therefore any cycles arising in the latter will definitely arise in the former.

Subsequently, we enable an analysis with full-tensor-precision only when no evaluation cycles have arisen. In this way, we can leverage higher precision in the common case of TensorFlow programs that do not employ recursion or looping at the Python level, instead delegating complex computation to library operators.

6 Discussion

Tensor shapes are reminiscent of types. It is, therefore, interesting to consider the relationship between our analysis and type checking, as well as the overall potential for a static type system for TensorFlow functionality.

Although the boundary between static analysis and type checking is not always clear, our static checker is best classified as a static analysis. Key factors in this classification are the whole-program and extensional nature of the analysis, as well as the intended soundness in reasoning.

Extensional Representation

The analysis represents value sets extensionally, i.e., by listing all their contents, instead of trying to abstract over them. For instance, if a tensor variable τ is inferred to hold possible shapes `[4, 3, 3, 2]`, `[None, 45]`, and `[30]`, the analysis will maintain the three different shape values explicitly, instead of trying to unify them in a single, more abstract, shape. This is a property more commonly found in static analyses than in type systems – the latter typically summarize values, at least at the level of program modules (e.g., functions). Static analyses also employ abstraction, but only do so based on the properties of the values themselves (e.g., when two values join in an abstract lattice).

Modular vs. Whole-Program

A type system typically emphasizes modular reasoning, forcing the summarization of values at the function boundary. In contrast, a context-sensitive whole-program static analysis will re-analyze a function under its different calling contexts. To maintain precision for different clients of a function, a type system employs polymorphism instead of context sensitivity: it expresses the type of a function in terms that may employ type variables, i.e., symbolic types that may assume multiple type values, instead of constants.

Sound vs. Best-Effort Reasoning

A static type system aims for soundness in certifying correct code, i.e., guarantees no false negatives. This implies that a type system has to be conservative in certifying correct code, yielding many false positive warnings. In contrast, a static analysis can strike any balance between true/false positive and true/false negative warnings as it deems appropriate for maximum usefulness.

TensorFlow Analysis

With the above factors in mind, it is interesting to consider the static checking of TensorFlow programs longer than toy examples. The example in Figure 1 is a slightly simplified version of one of the programs in the Zhang et al. [58] study. There is a bug in the last line of the program (the reshaping of `h_pool2`) which our analysis correctly warns about, but the main difficulty is in tracking shapes precisely in earlier program statements.

15:16 Static Analysis of Shape in TensorFlow Programs

```
import tensorflow as tf
def weight_variable(shape):
    initial = tf.truncated_normal(shape, stddev=0.1)
    return tf.Variable(initial)
def bias_variable(shape):
    initial = tf.constant(0.1, shape=shape)
    return tf.Variable(initial)
def conv2d(x, W):
    return tf.nn.conv2d(x, W, strides=[1, 1, 1, 1], padding='VALID')
def max_pool_2x2(x):
    return tf.nn.max_pool(x, ksize=[1, 2, 2, 1], strides=[1, 2, 2, 1], padding='SAME')

x = tf.placeholder(tf.float32, shape=[None, 784])

W_conv1 = weight_variable([5, 5, 1, 32])
b_conv1 = bias_variable([32])
x_image = tf.reshape(x, [-1, 28, 28, 1])
h_conv1 = tf.nn.relu(conv2d(x_image, W_conv1) + b_conv1)
h_pool1 = max_pool_2x2(h_conv1)

W_conv2 = weight_variable([5, 5, 32, 64])
b_conv2 = bias_variable([64])
h_conv2 = tf.nn.relu(conv2d(h_pool1, W_conv2) + b_conv2)
h_pool2 = max_pool_2x2(h_conv2)

W_fc1 = weight_variable([7 * 7 * 64, 1024])
b_fc1 = bias_variable([1024])
h_pool2_flat = tf.reshape(h_pool2, [-1, 7 * 7 * 64])
...
```

■ **Figure 1** Short but realistic example of the need for context sensitivity: program Unaligned-Tensor-1 (UT-1).

Context-sensitive reasoning is essential for this example. Functions `conv2d` (line 8) and `max_pool_2x2` (line 10) are each called twice (lines 18, 23, and 19, 24, respectively), each time with different shapes. (Pooling can be thought of as an operator analogous to convolution in terms of shape transformation.) Even for a small example such as this, a context-insensitive analysis would produce a highly imprecise result, with many false positives and negatives. A context-sensitive analysis considers, e.g., function `conv2d` twice, once for each argument shape, and can reason highly precisely about the effects of convolution on shape when all arguments (`x`, `W`, `strides`, as well as the padding strategy, `VALID`) have known values.

Conversely, consider what type signature (to capture shape) one might assign to function `conv2d` *modularly*, i.e., without knowing its arguments, `x` and `W`. Precise treatment of this function would require a polymorphic type system with considerable expressive power (e.g., integer arithmetic), as the modeling of Section 4 shows. This would more likely employ dependent typing, requiring significant human guidance for deciding interesting properties. It is interesting to further consider possible type signatures for a) the library method `tensorflow.nn.conv2d`, which has different shape behavior depending on the padding strategy; and b) library operations that employ broadcast. The current flexibility of these TensorFlow operators seems to require full algorithmic expressiveness (e.g., see our Datalog rules of Section 5.2.1) to capture well.

We conclude that the shape transformation of current TensorFlow operations requires a highly-expressive vocabulary, unlikely to be supported by a fully automatic type system. A statically-typed TensorFlow-like system would either require significant programmer assistance for sound reasoning, or curtail the flexibility of operators, to permit assigning them closed-form types.

7 Evaluation

Pythia runs at interactive speeds and has a Language Server Protocol integration with most popular IDEs. Therefore, although the analysis is applicable for development at any granularity, it mostly targets interactive feedback at development time. Our evaluation is set up accordingly.

Specifically, we evaluate the analysis against complete pre-existing programs from the recent study by Zhang et al. [58] on TensorFlow bugs. This gives us a curated dataset, collected independently, from real-world settings, and with ground truth relative to both the presence and the absence of bugs.⁴

The Zhang et al. study collects bugs from StackOverflow questions, and categorizes them based on their root causes and symptoms. One of the root cause categories is *Unaligned Tensor (UT)*, which maps exactly to shape violations detected by our analysis. There are 76 bugs that Zhang et al. manage to reproduce from StackOverflow questions⁵. Importantly, these 76 bugs map the entire, broad space of all TensorFlow bugs, most of which are out of the scope of our shape analysis. For instance, this includes low-accuracy computations, low-performance behavior, bugs related to API changes, and more.

There are 14 Unaligned Tensor (i.e., shape-related) bugs in the Zhang et al. study, and the study also provides fixed versions of the same programs, for a total of 28 test subjects, which form the universe set of our evaluation.

Executive Summary

With an 1-call-site-sensitive analysis and the *full-tensor-precision* option enabled, the analysis successfully detects 11 out of 14 bugs, with a single false positive repeated twice in the buggy and fixed version of UT5, for a precision of 84.62% (91.67% if repeat bugs are counted once) and recall of 78.6%. The average analysis time per program is under 1sec. Unless specified, the above analysis configuration is used. The effect of different analysis configurations on analysis precision is discussed in Section 7.3.

7.1 Classification of bugs

Table 1 summarizes the number of bugs reported. The bugs can be classified in the following categories:

- Operation Error: Tensor operation would throw a run-time error due to incompatible arguments provided.
- Incompatible fed data Error: Data fed to a placeholder tensor do not match the shape of the tensor.
- Broadcast/Reshape/Other Warning: Possibly confusing tensor operation behavior that would not cause a run-time error, as described in Section 5.

Table 2 serves as a detailed reference for each input program (including code URLs).

⁴ Links to all input programs can be found in Table 2. Pythia is part of the Doop repository (<https://bitbucket.org/yanniss/doop/>). A snapshot is contained in the artifact that accompanies this paper, together with detailed instructions for setting up and running Pythia.

⁵ The Zhang et al. study also collected a second dataset: 75 bugs from GitHub commits. We did not consider that dataset for reasons of engineering: a large number of these full Github programs use several external libraries, in addition to TensorFlow, as well as the full TensorFlow API. Modeling all of the required functionality, so that the potential of the approach is accurately evaluated, would require much more manpower than that of a research project. In contrast, the StackOverflow Zhang et al. benchmarks are well-isolated TensorFlow code patterns, which fit well the local, incremental nature of our approach and its implementation inside IDEs.

■ **Table 1** Detected bugs.

Bug type	Number of bugs
Operation Error	5
Incompatible fed data Error	2
Broadcast/Reshape/Other Warning	4
Total	11

7.2 Effectiveness and Efficiency

Overall, among the 14 input programs containing bugs, we successfully identify the bug in 11 programs. Our analysis produces a false positive in both the buggy and fixed versions of UT5 achieving 84.62% (11 of 13) overall *precision* – 100% for errors and 66.6% for warnings. The false positive appearing in both versions of UT5 is a warning for a reshape operation of a tensor with partially known shape into a shape with a special (-1) dimension, as described in section 5.2.2. In this case, the use of the reshape operation in a way we consider possibly confusing does not result in a bug.

The 3 false negatives produced by the analysis are a result of either API calls that we have not modeled or reliance on dynamic information to identify these bugs, leading to 78.6% *recall*. Of the 3 bugs our analysis could not detect, two (UT5 and UT10) are not detectable with static information alone. In both of them, the dataset is produced from information read from an external file. The code itself does not provide any hints about the shape of the dataset after it is read. As a result, the analysis cannot identify the incompatibility between the shape of the dataset and the tensor that will hold that data at run-time.

The analysis is compiled by the Soufflé [24] Datalog engine into an optimized C++ program and binary executable. The analysis code comprises several hundred non-trivial Datalog rules, therefore optimizing compilation is time-consuming, at 680sec. (All timings are from a single thread of a laptop with an Intel Core i7-3612QM 2.10GHz CPU, with 16GB of RAM.) Compilation is only performed once per analysis configuration, however, and the resulting analysis is highly efficient. For the input programs, the average analysis running time is just 0.26sec (median: 0.18sec, max: 0.49sec for UT4).

7.3 Precision

Pythia contains many precision enhancements – e.g., levels of context sensitivity, and a more detailed value abstraction. We already saw earlier, in Figure 1 an example of the impact of precision enhancements. We demonstrate the effect on the input programs of the evaluation set in Figure 2.

The figure shows seven input programs whose analysis precision changes for different configurations. (Input programs not shown either show no imprecision for any configuration or are the 3 for which our analysis misses the bug.) Precision is captured in three metrics: instances of imprecise tensor arguments (compared to the full achievable precision), false positives in analysis warnings, and instances of imprecise shapes. A check mark in the figure implies no imprecision for any metric. The four configurations of the analysis for each benchmark are:

- Configuration 1: context-insensitive (insens)
- Configuration 2: 1-call-site-sensitive (1call)
- Configuration 3: 1-call-site-sensitive + context-sensitive heap (1callH)
- Configuration 4: 1callH + full-tensor-precision.

■ **Table 2** “Unaligned Tensor (UT)” input programs (each entry corresponds to 2 programs: a fixed and a buggy version) and analysis reports. No UT14 exists in the input set. For the analyses reports, “—” designates an analysis terminating but reporting no bugs, “X” designates an analysis not terminating due to an exception. The URLs to access the programs are obtained by concatenating the following URL prefixes with the suffix for each program shown in the table’s second column. Github: <https://github.com/ForeverZyh/TensorFlow-Program-Bugs/blob/master/StackOverflow/> StackOverflow: <https://stackoverflow.com/q/>

Case Study	URLs	Description	Pythia Report	Ariadne Report
UT1	GitHub: UT-1/38167455-buggy/ mnist.py StackOverflow: 38167455	Reshape Operation	Warning	X
UT2	GitHub: UT-2/43067338-buggy/ multiplication.py StackOverflow: 43067338	matmul Incompatible Dimensions	Error	—
UT3	GitHub: UT-3/35451948-buggy/ image_set_shape.py StackOverflow: 35451948	Invalid call to <code>set_shape</code>	Error	—
UT4	GitHub: UT-4/44124668-buggy/ experiment.py StackOverflow: 44124668	Fed data don’t match shape	Error	X
UT5	GitHub: UT-5/43676638-buggy/ mnist.py StackOverflow: 43676638	Fed data don’t match shape	—	X
UT6	GitHub: UT-6/35295191-buggy/ word_representation.py StackOverflow: 35295191	matmul Incompatible Dimensions	Error	—
UT7	GitHub: UT-7/34079787-buggy/ playing.py StackOverflow: 34079787	Variable’s <code>initial_value</code> has unspecified shape	—	—
UT8	GitHub: UT-8/34908033-buggy/ multiply.py StackOverflow: 34079787	matmul Incompatible Dimensions	Error	X
UT9	GitHub: UT-9/40574552-buggy/ neural.py StackOverflow: 34908033	Incorrect operand shapes in <code>softmax_cross_entropy_with_logits</code>	Error	X
UT10	GitHub: UT-10/36343542-buggy/ tflin.py StackOverflow: 36343542	Fed data don’t match shape	—	X
UT11	GitHub: UT-11/41192992-buggy/ image.py StackOverflow: 41192992	Fed data don’t match shape	Error	—
UT12	GitHub: UT-12/43285733-buggy/ mnist.py StackOverflow: 43285733	Reshape Operation	Warning	X
UT13	GitHub: UT-12/42191656-buggy/ linear.py StackOverflow: 42191656	Misuse of <code>argmax</code> operation	Warning	—
UT15	GitHub: UT-15/38447935-buggy/ fitting.py StackOverflow: 38447935	Broadcasting operation	Warning	—

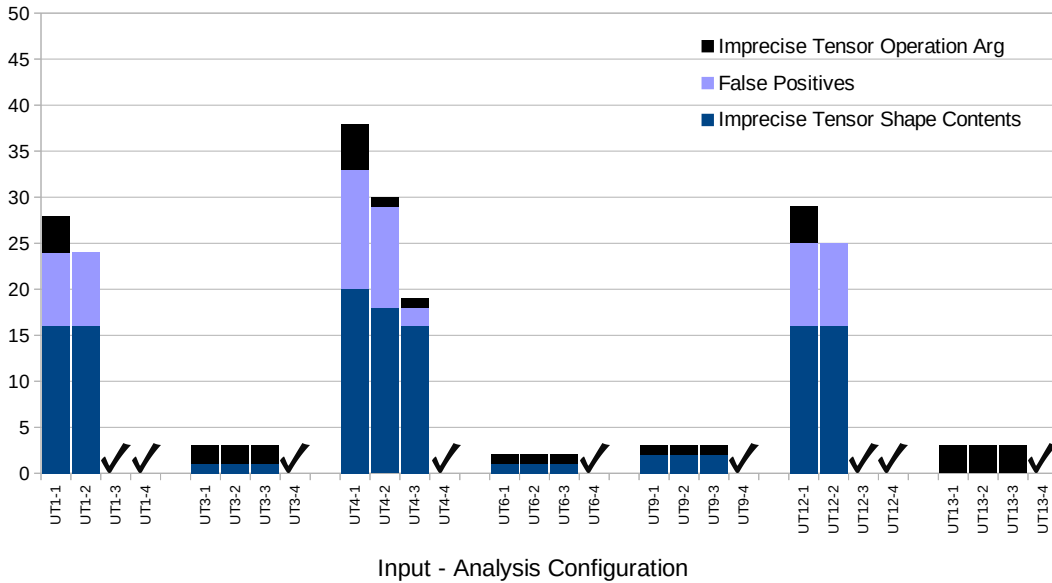


Figure 2 Chart: Imprecision metrics under different configurations. Lower is better, check mark is perfect precision. The y axis shows cumulative instances of three imprecision metrics: instances of imprecise tensor arguments, false positives in analysis warnings, and instances of imprecise shapes.

Among the programs presented in the chart we notice similar behavior for programs UT-3, UT-6, UT-9 and UT-13. For these and for the first 3 configurations, we notice minor imprecision but still no false positives. This is because the programs do not feature any calls to user-defined functions, but imprecision is introduced by other features, such as the use of `set_shape`. The introduction of full-tensor-precision removes any imprecisions.

The 3 remaining programs present large imprecision when using our less precise analysis configurations, resulting in many false positives. This is because, similar to the bug featured in Figure 1, the neural network is built using user-defined wrapper-functions, making context sensitivity necessary in order to achieve a highly-precise analysis. (These are also among the longest programs at around 100 or more lines.)

For instance, in UT-4,⁶ Pythia can correctly deduce the shapes of the tensors generated by two separate calls to function `generate_unit_test`, shown below: (This also showcases the analysis support for list comprehensions.)

```
def generate_unit_test(length):
    return [np.random.normal(0, 0.1, [56, 56, 3])
            for _ in range(length)],
            [random.randint(0, 9) for _ in range(length)]
```

In this input program, a false positive warning persists until the full-tensor-precision value abstraction is employed.

⁶ <https://github.com/ForeverZyh/TensorFlow-Program-Bugs/blob/master/StackOverflow/UT-4/44124668-buggy/experiment.py>

7.4 Other bugs found and missed

We next discuss selected cases of bugs reported or missed in the input dataset. Several interesting cases are already represented in our earlier examples, so we will not discuss them further. Namely, the reshape operation warning in case study UT1 has already been covered by the example of Section 6. The broadcasting operation warning in case study UT15 is analogous to Example 3 in Section 3. The incompatible dimensions error in `matmul`, appearing in case studies UT2, UT6 and UT8, is captured by Example 1 in Section 3.

Case Study UT3

Invalid call to `set_shape`.

```
import tensorflow as tf
import numpy as np

x = tf.placeholder(tf.float32, [None])
x.set_shape([1028178])
y = tf.identity(x)
y.set_shape([478, 717, 3])
X = np.random.normal(0, 0.1, 1028178)
```

The `set_shape` operation is used to provide additional, more concrete information about the shape of a tensor. In UT3, initially the shape of `x` is `[None]`. The first call to `set_shape` succeeds and establishes that the shape of `x` is `[1028178]`. The call to `identity` produces a tensor of the same shape as `x` and assigns it to `y`. The next call to `set_shape` is erroneous for two reasons. First, it attempts to specify an already established concrete shape. Secondly, even if the shape of `y` had not been already established by the first call to `set_shape`, the call would still fail since the dimensions of `[None]` and `[487, 717, 3]` are incompatible.

Case Study UT9

Incorrect operand shapes in `softmax_cross_entropy_with_logits` call.

```
import tensorflow as tf
import numpy as np
import random

n_feature = 10
n_data = 500
data = np.random.normal(0, 0.1, [n_data, n_feature])
label = [[random.randint(0, 1) for _ in range(n_data)]]

sizeOfRow = len(data[0])
x = tf.placeholder("float", shape=[None, sizeOfRow])
y = tf.placeholder("float")

prediction = neuralNetworkModel(x)
# using softmax function, normalize values to range(0,1)
error = tf.reduce_mean( tf.nn.softmax_cross_entropy_with_logits(
    logits=prediction, labels=y))
...
```

15:22 Static Analysis of Shape in TensorFlow Programs

In UT9 `softmax_cross_entropy_with_logits` is applied, which performs two operations. First, it applies the `softmax` function to denormalized log probabilities, i.e, the `logits` tensor, of shape `[batch_size, number_of_labels]`, to produce linear probabilities normalized to 1. It then computes the cross-entropy error of discrete classification based on the results of `softmax` and the ground truth classification of the training set that is held in the `labels` tensor, which is expected to have the same dimensions as `logits`. In this case, the shape of `logits` is `[500, 2]`, since there are 500 entries in the dataset and 2 labels (1 and 0), while the shape of `labels` is `[1,500]`.

Case Study UT7

Variable's initial value has unspecified shape. The bug in UT7 is representative of confusion regarding the TensorFlow execution model. It is also one that our analysis fails to capture, due to lack of modeling of the corresponding calls.

```
import tensorflow as tf
import random
import numpy as np

class Play:
    def __init__(self, input_data, labels):
        # the input shape is (batch_size, input_size)
        input_size = tf.shape(input_data)[1]

        # labels in one-hot format have shape (batch_size, num_classes)
        num_classes = tf.shape(labels)[1]
        stddev = 1.0 / tf.cast(input_size, tf.float32)

        w_shape = tf.stack([input_size, num_classes])
        normal_dist = tf.truncated_normal(w_shape, stddev=stddev, name='normaldist')
        self.w = tf.Variable(normal_dist, name='weights')
        print(self.w)

n_feature = 10
n_classes = 7
play = Play(tf.placeholder(tf.float32, [None, n_feature]),
            tf.placeholder(tf.int32, [None, n_classes]))
```

Recall from Section 2 that TensorFlow programs work by first setting up a data-flow pipeline of operators, and then executing it by feeding data. The Python code effectively *generates* a TensorFlow pipeline, before evaluating it. In case study UT7, the programmer incorrectly uses `tf.shape(input_data)` and `tf.shape(labels)`, while probably intending to use `input_data.get_shape()` and `labels.get_shape()`.

That is, the programmer intends to retrieve the shape of the dynamic data that will be fed into the TensorFlow pipeline. Instead, the erroneous calls retrieve the shape of the yet-unpopulated variables `input_data` and `labels`. The Python dynamic typing and TensorFlow tolerance conspire to propagate this error until it results in a shape mismatch later: each of the two erroneous calls returns an unevaluated one-dimensional tensor, which when dereferenced (via `[1]`) returns a to-be-evaluated integer. This integer is considered to be a zero-dimension tensor (`[]`), which becomes the value of `input_size` (and similarly for `num_classes`). TensorFlow then deduces that shape `w_shape` has value `[None, None]` as it is

the result of a `tf.stack` operation on two zero-dimension tensors, producing a 1-dimension tensor with shape `[2]`. However, the `tf.Variable` operation does not allow an unspecified shape as input, thus causing a crash.

Case Study UT11

Fed data don't match shape. The next input program indicates the handling of other operators (namely, `transpose`) as well as the ease with which a programmer can lose track of tensor shapes.

```

from tensorflow.contrib.keras.api.keras.preprocessing import image
import tensorflow as tf
import numpy as np

x = image.load_img(img_path, target_size=(250, 250))

x = image.img_to_array(x)
x_expanded = np.expand_dims(x, axis=0)
x_expanded_trans = np.transpose(x_expanded, [0, 3, 1, 2])

X = tf.placeholder(tf.float32, [None, 250, 250, 3])
sess = tf.Session()
sess.run(tf.global_variables_initializer())
print(sess.run(X, feed_dict={X: x_expanded_trans}))

```

In UT11, initially `x` is an image of shape `[250, 250, 3]` (because an image has 3 color channels). It is then converted to a NumPy array of the same shape, which is subsequently expanded to an array of shape `[1, 250, 250, 3]`, essentially creating a fake `batch_size` dimension. The array is then transposed to an array of shape `[1, 3, 250, 250]`. Finally, the code feeds the transposed array to a placeholder tensor of a different incompatible shape, `[None, 250, 250, 3]`, thus causing an error.

Case Study UT13

Misuse of `argmax` operation. Input program UT13 provides another example of a warning by our analysis that does not correspond to a run-time error, yet is highly likely to be a bug (as it is, in this case).

```

import tensorflow as tf

Y = tf.placeholder(tf.float32, shape=[4, 1], name='y')
...
Z = tf.argmax(Y, axis=1)
...

```

In UT13 `argmax` is applied to a tensor with shape `[4, 1]`. The `argmax` operation returns the index with the largest value along the specified `axis`. However, the second dimension of tensor `Y` is 1. In this case `argmax` returns a tensor with shape `[4]` with all 4 values being 0, since the *dimension size* in 1 is just 1. This is likely not the intended use of the `argmax` operation so we issue a warning, predicting that this promotion of values is not what the user aimed to accomplish.

7.5 Comparison with the state-of-the-art

The recent Ariadne tool [12] is, to our knowledge, the only static analysis tool that attempts to find shape bugs in TensorFlow code. We ran the latest version of Ariadne⁷ on our setup using the Language Server Protocol client for the Sublime text editor.

Table 2 shows the results of both tools for our dataset. The Ariadne tool reports 0 bugs. Furthermore, for half of the programs in our dataset, the Ariadne analysis ends with an exception, while for the other half it terminates successfully, reporting other information using the LSP protocol (such as call-site information) but no warning. These results can be explained by Ariadne’s limited support for tensor operations and by its not performing whole-program value-flow reasoning. For instance, Ariadne supports operators `reshape`, `set_shape`, `convolution`, and “node”, of which only `reshape` works fully. Pythia supports many more operations, such as `equal`, `add`, `multiply`, `matmul`, `argmax/argmin`, `transpose`, `expand_dims`, several pooling operations, and many shape pass-through operations. In Ariadne, tensors can be created using the `tf.placeholder` function. We also support `tf.constant`, `tf.Variable`, `tf.ones`, etc.

7.6 Threats to Validity

The largest threat is to external validity. Our findings may not generalize to other TensorFlow programs, especially of larger size. However, the benchmarks we examined are a prior and independently-identified set, collected from real-world reports. The programs are already large enough for context sensitivity and heap modeling to matter (as shown in Section 7.3).

8 Related Work

The space of checkers for machine learning programs is mostly populated by testing techniques [38, 51, 55]. Other approaches aid in the debugging and validation of machine learning programs. PALM [26] produces simplified decision-tree based meta models to facilitate the mapping of failed predictions to subsets of the training data. On the other hand LAMP [31] produces quantitative measurements that maps the impact of each input to each output in graph machine learning algorithms in an efficient way using partial derivatives. MODE [32] applies similar techniques for measuring the impact of each feature in Neural Networks.

The recent Ariadne tool [12] demonstrates an application of static analysis technology to TensorFlow, but neither models many TensorFlow operators, nor performs whole-program value-flow reasoning. This limits Ariadne’s applicability to artificial examples, with manually-planted bugs, and to Python input programs of very limited form – e.g., as discussed in Section 7.5 and shown in table 2, the system cannot run or produce useful results on *any* of our input benchmarks.

General program analysis tools for Python have been developed. These mostly aim to find type errors. Invariably, such frameworks restrict the features of Python, since the language is highly dynamic and its full static analysis with good precision is impossible. For instance, even determining which file is imported when an `import` statement is executed can be undecidable. RPython [5] is a statically typed subset of the Python language designed for writing partially evaluated interpreters. All metaprogramming features (including `eval` and metaclasses) may be used during the initialization of the Python classes. RPython is best compared with a statically typed version of Python. Retrofitting type systems to dynamic languages is a fairly common strategy, and examples include preemptive type

⁷ Downloaded from the official site: <https://wala.github.io/IDE/>.

checking for Python [16], DRuby for Ruby [14] or a type system for Erlang [34]. JavaScript has probably attracted the most attention in this space and there are many more examples of type systems for it [10, 11, 19, 22, 29]. These systems are used either for speeding up JavaScript implementations or for type checking during development. Due to the complexity of the underlying problem, many authors (including ourselves) have found it more fruitful to concentrate on type checking or bug finding for specific domains. Related examples in the wild include a system [4] for Ruby on Rails or the work of [28] for static detection of JQuery errors in JavaScript by identifying inconsistencies between the actual page structure and query expectations.

The space of static analysis tools for Python is relatively sparse. Python Taint [45] is a static analysis tool for detecting security vulnerabilities. It uses standard data-flow techniques, and can do some interprocedural analysis. However, its interprocedural reasoning is limited: it looks for a definition of a function for a call using its name, rather than handling function pointers and object semantics, as needed even for simple realistic examples.

Gorbovitski et al. [15] developed a context-sensitive, flow-sensitive alias analysis for Python for program optimization. They offer several significant insights on the precision needed for dynamic languages. The analysis appears sophisticated but we have not found an available implementation for reuse.

Other tools are shallow code quality checkers or lint tools; examples are Pylint [44], pycodestyle [41], pyflakes [43], Flake8 [13], pydocstyle [42], jedi [21], bandit [7] and mccabe [35]. Prospector [40] combines several of these tools. These tools are all local analyses, for instance, mccabe focuses on the syntactic code complexity of single functions and others focus on code style issues.

Another bug detection approach includes analyses that are dynamic, yet generalize from concrete executions. Xu et al. [56] developed such a predictive analysis for Python, detecting more general bugs, such as Attribute Errors and Type Errors, and Unicode Encode/Decode Errors which are specific to web applications.

Finally, although the work we describe is applied to TensorFlow, the principles described may apply to other scientific computing languages and extensions such as SAC [17] or LAPACK [6].

9 Conclusions

We presented a static analysis approach for detecting shape bugs in TensorFlow programs. The analysis models value-flow in Python programs and closely tracks the rich shape-transformation semantics of TensorFlow operators. The result is the first concrete demonstration of the applicability of static analysis for detecting realistic bugs in the TensorFlow domain. The analysis is highly efficient and very effective over an independently-collected set of input programs that sample the universe of real-world TensorFlow bugs.

References

- 1 Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org. URL: <https://www.tensorflow.org/>.

- 2 Rami Al-Rfou, Guillaume Alain, Amjad Almahairi, Christof Angermueller, Dzmitry Bahdanau, Nicolas Ballas, Frederic Bastien, Justin Bayer, Anatoly Belikov, and others . Theano: A python framework for fast computation of mathematical expressions. *arXiv preprint*, January 2016. [arXiv:1605.02688](https://arxiv.org/abs/1605.02688).
- 3 Miltiadis Allamanis, Earl Barr, Premkumar Devanbu, and Charles Sutton. A survey of machine learning for big code and naturalness. *ACM Computing Surveys*, 51, September 2017. [doi:10.1145/3212695](https://doi.org/10.1145/3212695).
- 4 Jong-hoon An, Avik Chaudhuri, and Jeffrey S. Foster. Static Typing for Ruby on Rails. In *Proceedings of ASE*, pages 590–594, November 2009. [doi:10.1109/ASE.2009.80](https://doi.org/10.1109/ASE.2009.80).
- 5 Davide Ancona, Massimo Ancona, Antonio Cuni, and Nicholas D. Matsakis. Rpython: A step towards reconciling dynamically and statically typed oo languages. In *Proceedings of the 2007 Symposium on Dynamic Languages*, DLS '07, pages 53–64, New York, NY, USA, 2007. ACM. [doi:10.1145/1297081.1297091](https://doi.org/10.1145/1297081.1297091).
- 6 Ed Anderson, Zhaojun Bai, Jack Dongarra, A. Greenbaum, A. McKenney, Jeremy Du Croz, Sven Hammarling, James Demmel, Christian Bischof, and Danny C. Sorensen. Lapack: A portable linear algebra library for high-performance computers. In *Proceedings of the 1990 ACM/IEEE Conference on Supercomputing*, Supercomputing '90, pages 2–11, Washington, DC, USA, 1990. IEEE Computer Society Press.
- 7 bandit. <https://github.com/openstack/bandit>. Accessed: 2020-01-06.
- 8 Martin Bravenboer and Yannis Smaragdakis. Exception analysis and points-to analysis: Better together. In *Proc. of the 18th International Symp. on Software Testing and Analysis*, ISSTA '09, pages 1–12, New York, NY, USA, 2009. ACM. [doi:10.1145/1572272.1572274](https://doi.org/10.1145/1572272.1572274).
- 9 Martin Bravenboer and Yannis Smaragdakis. Strictly declarative specification of sophisticated points-to analyses. In *Proc. of the 24th Annual ACM SIGPLAN Conf. on Object Oriented Programming, Systems, Languages, and Applications*, OOPSLA '09, New York, NY, USA, 2009. ACM.
- 10 Satish Chandra, Colin S. Gordon, Jean-Baptiste Jeannin, Cole Schlesinger, Manu Sridharan, Frank Tip, and Youngil Choi. Type inference for static compilation of JavaScript. *SIGPLAN Not.*, 51(10):410–429, October 2016. [doi:10.1145/3022671.2984017](https://doi.org/10.1145/3022671.2984017).
- 11 Wontae Choi, Satish Chandra, George C. Necula, and Koushik Sen. Sjs: A type system for JavaScript with fixed object layout. In Sandrine Blazy and Thomas Jensen, editors, *SAS*, volume 9291 of *Lecture Notes in Computer Science*, pages 181–198. Springer, 2015. URL: <http://dblp.uni-trier.de/db/conf/sas/sas2015.html#ChoiCNS15>.
- 12 Julian Dolby, Avraham Shinnar, Allison Allain, and Jenna Reinen. Ariadne: Analysis for machine learning programs. In *Proceedings of the 2Nd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*, MAPL 2018, pages 1–10, New York, NY, USA, 2018. ACM. [doi:10.1145/3211346.3211349](https://doi.org/10.1145/3211346.3211349).
- 13 Flake8. <https://github.com/PyCQA/flake8>. Accessed: 2020-01-06.
- 14 Michael Furr, Jong-hoon (David) An, and Jeffrey S. Foster. Profile-guided static typing for dynamic scripting languages. In *Proceedings of OOPSLA*, pages 283–300, 2009. [doi:10.1145/1639949.1640110](https://doi.org/10.1145/1639949.1640110).
- 15 Michael Gorbovitski, Yanhong A. Liu, Scott D. Stoller, Tom Rothamel, and Tuncay K. Tekle. Alias analysis for optimization of dynamic languages. In *Proceedings of the 6th Symposium on Dynamic Languages*, DLS '10, pages 27–42, New York, NY, USA, 2010. ACM. [doi:10.1145/1869631.1869635](https://doi.org/10.1145/1869631.1869635).
- 16 Neville Grech, Bernd Fischer, and Julian Rathke. Preemptive type checking. *Journal of Logical and Algebraic Methods in Programming*, 101:151–181, 2018. [doi:10.1016/j.jlamp.2018.08.003](https://doi.org/10.1016/j.jlamp.2018.08.003).
- 17 Clemens Grelck and Sven-Bodo Scholz. SAC – a functional array language for efficient multi-threaded execution. *International Journal of Parallel Programming*, 34(4):383–427, August 2006. [doi:10.1007/s10766-006-0018-x](https://doi.org/10.1007/s10766-006-0018-x).

- 18 Salvatore Guarnieri and Benjamin Livshits. GateKeeper: mostly static enforcement of security and reliability policies for Javascript code. In *Proc. of the 18th USENIX Security Symposium*, SSYM'09, pages 151–168, Berkeley, CA, USA, 2009. USENIX Association. URL: <http://dl.acm.org/citation.cfm?id=1855768.1855778>.
- 19 Brian Hackett and Shu-yu Guo. Fast and precise hybrid type inference for JavaScript. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '12, pages 239–250, New York, NY, USA, 2012. ACM. doi:10.1145/2254064.2254094.
- 20 Elnar Hajiyev, Mathieu Verbaere, and Oege de Moor. CodeQuest: Scalable source code queries with Datalog. In *Proc. of the 20th European Conf. on Object-Oriented Programming*, ECOOP '06, pages 2–27. Springer, 2006.
- 21 jedi. <https://github.com/davidhalter/jedi>. Accessed: 2020-01-06.
- 22 Simon Holm Jensen, Anders Möller, and Peter Thiemann. Type analysis for JavaScript. In *Proceedings of the 16th International Symposium on Static Analysis*, SAS '09, pages 238–255, Berlin, Heidelberg, 2009. Springer-Verlag. doi:10.1007/978-3-642-03237-0_17.
- 23 Melvin Johnson, Mike Schuster, Quoc V. Le, Maxim Krikun, Yonghui Wu, Zhifeng Chen, Nikhil Thorat, Fernanda Viégas, Martin Wattenberg, Greg Corrado, Macduff Hughes, and Jeffrey Dean. Google's multilingual neural machine translation system: Enabling zero-shot translation. *Transactions of the Association for Computational Linguistics*, 5:339–351, 2017. URL: <https://transacl.org/ojs/index.php/tac1/article/view/1081>.
- 24 Herbert Jordan, Bernhard Scholz, and Pavle Subotić. Soufflé: On synthesis of program analyzers. In Swarat Chaudhuri and Azadeh Farzan, editors, *Computer Aided Verification*, pages 422–430, Cham, 2016. Springer International Publishing.
- 25 George Kastrinis and Yannis Smaragdakis. Hybrid context-sensitivity for points-to analysis. In *Proc. of the 2013 ACM SIGPLAN Conf. on Programming Language Design and Implementation*, PLDI '13, New York, NY, USA, 2013. ACM.
- 26 Sanjay Krishnan and Eugene Wu. Palm: Machine learning explanations for iterative debugging. In *Proceedings of the 2Nd Workshop on Human-In-the-Loop Data Analytics*, HILDA'17, pages 4:1–4:6, New York, NY, USA, 2017. ACM. doi:10.1145/3077257.3077271.
- 27 Monica S. Lam, John Whaley, V. Benjamin Livshits, Michael C. Martin, Dzintars Avots, Michael Carbin, and Christopher Unkel. Context-sensitive program analysis as database queries. In *Proc. of the 24th Symp. on Principles of Database Systems*, PODS '05, pages 1–12, New York, NY, USA, 2005. ACM. doi:10.1145/1065167.1065169.
- 28 Benjamin S. Lerner, Liam Elberty, Jincheng Li, and Shriram Krishnamurthi. Combining form and function: Static types for jquery programs. In *Proceedings of the 27th European Conference on Object-Oriented Programming*, ECOOP'13, pages 79–103, Berlin, Heidelberg, 2013. Springer-Verlag. doi:10.1007/978-3-642-39038-8_4.
- 29 Benjamin S. Lerner, Joe Gibbs Politz, Arjun Guha, and Shriram Krishnamurthi. Tejas: Retrofitting type systems for JavaScript. *SIGPLAN Not.*, 49(2):1–16, October 2013. doi:10.1145/2578856.2508170.
- 30 Percy Liang and Mayur Naik. Scaling abstraction refinement via pruning. In *Proc. of the 2011 ACM SIGPLAN Conf. on Programming Language Design and Implementation*, PLDI '11, pages 590–601, New York, NY, USA, 2011. ACM. doi:10.1145/1993498.1993567.
- 31 Shiqing Ma, Yousra Aafer, Zhaogui Xu, Wen-Chuan Lee, Juan Zhai, Yingqi Liu, and Xiangyu Zhang. Lamp: Data provenance for graph based machine learning algorithms through derivative computation. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2017, pages 786–797, New York, NY, USA, 2017. ACM. doi:10.1145/3106237.3106291.
- 32 Shiqing Ma, Yingqi Liu, Wen-Chuan Lee, Xiangyu Zhang, and Ananth Grama. Mode: Automated neural network model debugging via state differential analysis and input selection. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2018, pages 175–186, New York, NY, USA, 2018. ACM. doi:10.1145/3236024.3236082.

- 33 Magnus Madsen, Benjamin Livshits, and Michael Fanning. Practical static analysis of JavaScript applications in the presence of frameworks and libraries. In *Proc. of the ACM SIGSOFT International Symp. on the Foundations of Software Engineering, FSE '13*, pages 499–509. ACM, 2013. doi:10.1145/2491411.2491417.
- 34 Simon Marlow and Philip Wadler. A practical subtyping system for Erlang. In *Proceedings of ICFP*, pages 136–149, August 1997. doi:10.1145/258949.258962.
- 35 mccabe. <https://pypi.python.org/pypi/mccabe>. Accessed: 2020-01-06.
- 36 Mayur Naik, Alex Aiken, and John Whaley. Effective static race detection for java. In *Proc. of the 2006 ACM SIGPLAN Conf. on Programming Language Design and Implementation, PLDI '06*, pages 308–319, New York, NY, USA, 2006. ACM. doi:10.1145/1133981.1134018.
- 37 Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. Automatic differentiation in pytorch. In *NIPS-W*, 2017.
- 38 Kexin Pei, Yinzhi Cao, Junfeng Yang, and Suman Jana. Deepxplore: Automated whitebox testing of deep learning systems. In *Proceedings of the 26th Symposium on Operating Systems Principles, SOSP '17*, pages 1–18, New York, NY, USA, 2017. ACM. doi:10.1145/3132747.3132785.
- 39 Simon J. D. Prince. *Computer Vision: Models, Learning, and Inference*. Cambridge University Press, New York, NY, USA, 1st edition, 2012.
- 40 Prospector. <https://prospector.readthedocs.io/en/master/>. Accessed: 2020-01-06.
- 41 pycodestyle. <http://pep8.readthedocs.org/en/latest/>. Accessed: 2020-01-06.
- 42 pydocstyle. <https://github.com/PyCQA/pydocstyle>. Accessed: 2020-01-06.
- 43 pyflakes. <https://launchpad.net/pyflakes>. Accessed: 2020-01-06.
- 44 Pylint. <http://www.pylint.org/>. Accessed: 2020-01-06.
- 45 Python Taint. <https://github.com/python-security/pyt>. Accessed: 2020-01-06.
- 46 Thomas W. Reps. Demand interprocedural program analysis using logic databases. In R. Ramakrishnan, editor, *Applications of Logic Databases*, pages 163–196. Kluwer Academic Publishers, 1994.
- 47 Micha Sharir and Amir Pnueli. Two approaches to interprocedural data flow analysis. In Steven S. Muchnick and Neil D. Jones, editors, *Program flow analysis: theory and applications*, chapter 7, pages 189–233. Prentice-Hall, Inc., Englewood Cliffs, NJ, 1981.
- 48 Olin Shivers. *Control-Flow Analysis of Higher-Order Languages*. PhD thesis, Carnegie Mellon University, may 1991.
- 49 Yannis Smaragdakis and George Balatsouras. Pointer analysis. *Foundations and Trends in Programming Languages*, 2(1):1–69, 2015. doi:10.1561/25000000014.
- 50 Manu Sridharan, Satish Chandra, Julian Dolby, Stephen J. Fink, and Eran Yahav. Alias analysis for object-oriented programs. In Dave Clarke, James Noble, and Tobias Wrigstad, editors, *Aliasing in Object-Oriented Programming. Types, Analysis and Verification*, volume 7850 of *Lecture Notes in Computer Science*, pages 196–232. Springer Berlin Heidelberg, 2013. doi:10.1007/978-3-642-36946-9_8.
- 51 Yuchi Tian, Kexin Pei, Suman Jana, and Baishakhi Ray. Deeptest: Automated testing of deep-neural-network-driven autonomous cars. In *Proceedings of the 40th International Conference on Software Engineering, ICSE '18*, pages 303–314, New York, NY, USA, 2018. ACM. doi:10.1145/3180155.3180220.
- 52 Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie J. Hendren, Patrick Lam, and Vijay Sundaresan. Soot - a Java bytecode optimization framework. In *Proc. of the 1999 Conf. of the Centre for Advanced Studies on Collaborative research, CASCON '99*, pages 125–135. IBM Press, 1999. URL: <http://dl.acm.org/citation.cfm?id=781995.782008>.
- 53 John Whaley, Dzintars Avots, Michael Carbin, and Monica S. Lam. Using Datalog with binary decision diagrams for program analysis. In *Proc. of the 3rd Asian Symp. on Programming Languages and Systems*, pages 97–118. Springer, 2005. doi:10.1007/11575467_8.

- 54 John Whaley and Monica S. Lam. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In *Proc. of the 2004 ACM SIGPLAN Conf. on Programming Language Design and Implementation*, PLDI '04, pages 131–144, New York, NY, USA, 2004. ACM. doi:10.1145/996841.996859.
- 55 Xiaoyuan Xie, Joshua W. K. Ho, Christian Murphy, Gail Kaiser, Baowen Xu, and Tsong Yueh Chen. Testing and validating machine learning classifiers by metamorphic testing. *J. Syst. Softw.*, 84(4):544–558, April 2011. doi:10.1016/j.jss.2010.11.920.
- 56 Zhaogui Xu, Peng Liu, Xiangyu Zhang, and Baowen Xu. Python predictive analysis for bug detection. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2016, pages 121–132, New York, NY, USA, 2016. ACM. doi:10.1145/2950290.2950357.
- 57 Xin Zhang, Ravi Mangal, Radu Grigore, Mayur Naik, and Hongseok Yang. On abstraction refinement for program analyses in Datalog. In *Proc. of the 2014 ACM SIGPLAN Conf. on Programming Language Design and Implementation*, PLDI '14, pages 239–248, New York, NY, USA, 2014. ACM. doi:10.1145/2594291.2594327.
- 58 Yuhao Zhang, Yifan Chen, Shing-Chi Cheung, Yingfei Xiong, and Lu Zhang. An empirical study on tensorflow program bugs. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2018, pages 129–140, New York, NY, USA, 2018. ACM. doi:10.1145/3213846.3213866.

Value Partitioning: A Lightweight Approach to Relational Static Analysis for JavaScript

Benjamin Barslev Nielsen

Aarhus University, Denmark
barslev@cs.au.dk

Anders Møller 

Aarhus University, Denmark
amoeller@cs.au.dk

Abstract

In static analysis of modern JavaScript libraries, relational analysis at key locations is critical to provide sound and useful results. Prior work addresses this challenge by the use of various forms of trace partitioning and syntactic patterns, which is fragile and does not scale well, or by incorporating complex backwards analysis. In this paper, we propose a new lightweight variant of trace partitioning named value partitioning that refines individual abstract values instead of entire abstract states. We describe how this approach can effectively capture important relational properties involving dynamic property accesses, functions with free variables, and predicate functions. Furthermore, we extend an existing JavaScript analyzer with value partitioning and demonstrate experimentally that it is a simple, precise, and efficient alternative to the existing approaches for analyzing widely used JavaScript libraries.

2012 ACM Subject Classification Theory of computation → Program analysis

Keywords and phrases JavaScript, dataflow analysis, abstract interpretation

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2020.16

Funding This work was supported by the European Research Council (ERC) under the European Union’s Horizon 2020 research and innovation program (grant agreement No 647544).

1 Introduction

JavaScript programs are challenging to analyze statically due to the dynamic nature of the language. One of the main obstacles is the presence of *dynamic property access* operations that allow objects to be manipulated using object property names that are dynamically computed strings. A typical pattern that has received much attention is *correlated read/write pairs* [25], a simple variant of which looks as follows:

$$t = x[p]; \quad \dots \quad y[p] = t;$$

At run-time, this code copies a property whose name is the value of p from the x object to the y object. If the static analysis does not know precisely the string value of p , then the properties of x will be mixed together in y . Experience with analyzers such as WALA [25, 24, 28], SAFE [17, 22], JSAI [13], and TAJIS [11, 2, 26] has shown that when analyzing real-world JavaScript code, including jQuery, Lodash, Underscore and other widely used libraries, such situations often cause an avalanche of spurious dataflow that makes the analysis results useless. If, for example, x is the object $\{m1: f1, m2: f2, \dots, m10: f10\}$ where $f1, f2, \dots, f10$ are functions, then any subsequent function call, for example $y.m3(\dots)$, will be treated by the analysis as a call to any of the 10 functions.

Several analysis techniques have been proposed to address this challenge. The techniques based on correlation tracking [25], static/dynamic determinacy [24, 2], and loop sensitivity [22] aim to increase precision by the use of context sensitivity or loop unrolling to ensure that



© Benjamin Barslev Nielsen and Anders Møller;

licensed under Creative Commons License CC-BY

34th European Conference on Object-Oriented Programming (ECOOP 2020).

Editors: Robert Hirschfeld and Tobias Pape; Article No. 16; pp. 16:1–16:28

Leibniz International Proceedings in Informatics



LIPICs Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany



the analysis has precise information about \mathbf{p} in the example above. Although this approach works well in many cases, the aggressive use of context sensitivity or loop unrolling can be expensive on analysis time. Even more importantly, it falls short when \mathbf{p} is not determinate (i.e., when its value is not fixed even when the call context is known).

An important step forward is the approach used in the CompAbs analyzer, which is built on SAFE [16]: Even if \mathbf{p} is imprecise, the loss of precision at the property write operation can be avoided by applying trace partitioning [23] at the property read operation, based on which properties exist on \mathbf{x} . Intuitively, it is often not necessary to have precise information about \mathbf{p} ; instead we can refine the current abstract state into a collection of more precise partitions, one for each of the 10 properties of \mathbf{x} (plus one extra for the case where \mathbf{p} is none of those strings, but let us ignore that for now), and after the property write operation merge them again. (The same idea was used earlier in TAJs, but only at **for-in** loops, not at dynamic property reads [2].) This approach, however, also has drawbacks. Trace partitioning is expensive, so it must be used scarcely: in the example, the code between the dynamic property read and the dynamic property write is essentially analyzed 10 times. For this reason, CompAbs relies on a syntactic pre-analysis to recognize different kinds of correlated read/write pairs for guiding the creation and merging of partitions.

Recent work [26] has shown that the syntactic pre-analysis approach of CompAbs is too fragile, for example, it is incapable of analyzing the Lodash library (see Section 2), and demand-driven value refinement has been proposed as an alternative. Instead of relying on context sensitivity, loop unrolling, or trace partitioning, that approach applies, during the analysis when encountering a dynamic property write operation with an imprecise property name, a separate backwards analysis to regain the relation between the property name and the value to be written. Although demand-driven value refinement has been shown to work quite well in practice, building a backwards analysis for the full JavaScript language and its standard library is a major endeavor, so developing simpler alternatives is desirable.

Our approach builds upon the observation from CompAbs that sufficient precision can be obtained using trace partitioning based on the properties of the object being read. Our key insight is that we do not need to partition the entire abstract state as done by CompAbs: It suffices to only partition the abstract values for the property name \mathbf{p} and the value being read $\mathbf{x}[\mathbf{p}]$ in the above example. This means that instead of analyzing the code 10 times, we only analyze it once, but using partitioned abstract values that retain the correlation between \mathbf{p} and $\mathbf{x}[\mathbf{p}]$. The partitioned abstract values are introduced at $\mathbf{t} = \mathbf{x}[\mathbf{p}]$ and used at $\mathbf{y}[\mathbf{p}] = \mathbf{t}$ by means of specialized transfer functions. We refer to this variant of trace partitioning as *value partitioning*. Since partitioning individual abstract values does not increase the analysis complexity as much as partitioning entire states, it becomes feasible to apply value partitioning more extensively, at every dynamic property read where the property name is imprecise, thereby obviating the need for the syntactic pre-analysis.

In this paper we present a theoretical framework for value partitioning, together with three instantiations: *property-name partitioning* (which is the one used in the example above), *free-variable partitioning* (to improve precision for free variables of closures), and *type partitioning* (to improve precision for predicate functions). Additionally, we extend the static analyzer TAJs with all three kinds of value partitioning and demonstrate that the approach is effective for analyzing popular JavaScript libraries. Value partitioning is a lightweight alternative to the existing approaches to relational analysis for JavaScript: Compared to CompAbs-style trace partitioning it avoids many redundant computations caused by similarities between different partitions, and compared to demand-driven value refinement it avoids the need for creating a separate backwards analysis.

```

1 function mixin(object, source) {
2   baseFor(source, function (func, methodName) {
3     if (!isFunction(func))
4       return;
5     object[methodName] = func;
6     if (isFunction(object))
7       object.prototype[methodName] = function() {
8         ...
9         func.apply(...);
10      }
11   });
12 }
13
14 function baseFor(source, iteratee) {
15   Object.keys(source).forEach(function (key) {
16     iteratee(source[key], key);
17   });
18 }
19
20 // usage of mixin during initialization
21 mixin(lodash, lodash);

```

■ **Figure 1** Motivating example based on code from the Lodash library.

In summary our contributions are:

- Value partitioning: a general static analysis technique that is capable of reasoning about relations between abstract values.
- Three instantiations of value partitioning, which tackle different challenges in static analysis for JavaScript, each involving relational properties:
 - property-name partitioning: relations between dynamically computed object property names and values;
 - free-variable partitioning: relations between functions and their free variables; and
 - type partitioning: relations between arguments and return values of predicate functions.
- Experimental results: We show that value partitioning makes TAJs more precise than CompAbs [16] for several real-world JavaScript libraries, including Lodash, which is the most widely used library. The resulting precision is comparable to (and in case of the Lodash4 benchmark group substantially higher than) that of demand-driven value refinement [26], without the need for a separate backwards analysis.

2 Motivating Example and Overview

Figure 1 shows a small code example based on Lodash (version 4.17.10), which is the most depended-upon of all npm packages.¹ Lines 1–12 define the function `mixin`, which copies all function properties from `source` to `object`. If `object` is a function, a new function (which

¹ Lodash (<https://lodash.com/>) has more than 115 000 dependents in npm and more than 27 million weekly downloads as of May 2020.

16:4 Value Partitioning

on invocation calls the function to be copied) is also copied to `object.prototype`, such that instantiations of `object` (using the keyword `new`) also will have these functions. In line 21, which is executed during the initialization of `Lodash`, `mixin` is called with the library object as both arguments. The function `mixin` uses a helper function `baseFor` defined in lines 14–18. It is called with `source` and a callback function defined in lines 2–11. The `baseFor` function then gets all the object property names from the `source` object using `Object.keys`, and the callback function is called (line 16) for each property name and corresponding property value. Line 3 checks whether `func` is a function. If so, the function is copied to `object[methodName]` in line 5. Note that `func` actually is the value `source[methodName]`. Line 6 checks whether `object` is also a function and if so, a new function is declared and written to `object.prototype[methodName]` in line 7. When invoked, that new function calls `func` using `func.apply(...)` in line 9.

Such complex code is not unusual in modern JavaScript libraries. For a static analysis reasoning about the dataflow in this code, the correlation between `methodName` and `func` is critical. An analysis that loses track of this correlation will mix together all the properties of the library object `lodash` when analyzing the call `mixin(lodash, lodash)` in line 21. As a consequence, if the program being analyzed contains a call to, for example, `lodash.map`, that will be treated by the analysis as a call to any of `Lodash`'s more than 100 different functions, not only the actual `map` function, thereby triggering an avalanche of spurious dataflow.

Existing approaches

Existing JavaScript analyzers do not have precise information about the value of `key` in line 16, for various different reasons. (Most importantly, `Object.keys` produces an array of property names in unspecified order.) Previous work has suggested two approaches to analyze such code precisely even when `key` is imprecise. The `CompAbs` [16] approach uses trace partitioning guided by syntactic patterns. If trace partitioning is used at the dynamic property read operation in line 16, the abstract state is partitioned into a set of refined abstract states corresponding to the properties of the `source` object. This way the value of `key` is precise in each of those states, and the call in line 16 is analyzed separately for each of them. Trace partitioning, however, is expensive, so `CompAbs` limits the use of trace partitioning according to certain syntactic patterns. At this specific dynamic property operation, `CompAbs` chooses not to apply trace partitioning and fails to detect that the relation between `methodName` and `func` is important.

The second approach is demand-driven value refinement [26], which can analyze the example code with sufficient precision to avoid mixing together the `Lodash` functions. With this approach, the analysis detects imprecision at the dynamic property write in line 5: `methodName` is an imprecise string and `func` can be many different functions. It then queries a backwards abstract interpreter asking for the possible value of `methodName` for each of the functions. The backwards analysis returns a precise property name for each function and thereby enables the dynamic property write operation to be modeled precisely. For the dynamic property write in line 7, the function defined in lines 7–10 is written to all properties of `object.prototype`, but the abstract value being written is augmented, such that the value of `methodName` remains precise. When reading `func` in line 9, the backwards analysis is queried to get the value of `func` relative to the value of `methodName`, thereby retrieving a precise value for `func`. This ensures the desired precision, but the approach requires a complicated backwards analysis.

Value partitioning

We will now informally explain how value partitioning can provide similar precision as demand-driven value refinement, but without the need for a backwards abstract interpreter. With traditional trace partitioning, as used by, for example, CompAbs, the analysis can track multiple abstract states for each program point, such that the different abstract states cover different assumptions about the execution paths that lead to that point. (Correlation tracking [25], determinacy-based analysis [24, 2], and loop sensitivity [22] can also be viewed as variations of trace partitioning.) The key idea behind value partitioning is that we can obtain a similar effect as trace partitioning by instead performing the partitioning at the level of individual abstract values. In principle, the resulting abstract domain is isomorphic to a traditional trace partitioning domain, but this approach provides more flexibility for using different kinds of partitioning for different parts of the abstract states. This general idea can be instantiated in multiple ways to track different kinds of relational properties. We next describe three instantiations that enable precise analysis of challenging JavaScript code, including the Lodash example.

Property name partitioning

One instantiation is property name partitioning, which performs partitioning at dynamic property reads, similar to the CompAbs technique, but on abstract values instead of abstract states. To illustrate this mechanism by example, consider the read operation in line 16 and the correlated write operation in line 5. Assume for simplicity that the `source` object has only two properties, `{map: f1, trim: f2}` where `f1` and `f2` are functions, and `methodName` is an abstract value that overapproximates all valid property names. When reading `source[methodName]`, an analysis without value partitioning will read all the properties of `source`. When using value partitioning, we instead partition this value according to the property names of `source`, meaning that we obtain a value $[t_1 \mapsto \mathbf{f1}, t_2 \mapsto \mathbf{f2}, t_3 \mapsto \mathbf{undefined}]$ where t_1 , t_2 , and t_3 represent different partitions.² Intuitively, t_1 represents the execution traces where the property name being read is `map`, t_2 similarly represents traces where the property name being read is `trim`, and t_3 represents all other traces. We similarly write the partitioned value $[t_1 \mapsto \mathbf{"map"}, t_2 \mapsto \mathbf{"trim"}, t_3 \mapsto \mathbf{AnyString}]$ to `methodName`.³ In this way, the resulting abstract state retains the correlation between the values of `methodName` and `source[methodName]`.

Later the analysis reaches the write operation `object[methodName] = func`, with an abstract state where `methodName` is $[t_1 \mapsto \mathbf{"map"}, t_2 \mapsto \mathbf{"trim"}, t_3 \mapsto \mathbf{AnyString}]$ and `func` is $[t_1 \mapsto \mathbf{f1}, t_2 \mapsto \mathbf{f2}, t_3 \mapsto \mathbf{undefined}]$. Since the property name and the value to be written have the same partitions, we can perform the dynamic property write separately for each partition, meaning that `f1` is written to the `map` property, and analogously for the other two partitions, thereby avoiding mixing together the properties.

Since the partitioning is performed at the value level, unlike traditional trace partitioning we do not need any extra call contexts to the callback function defined in line 2, so the overhead of value partitioning is negligible, even when the correlated read/write pairs span multiple functions. For this reason, we can apply property name partitioning at all dynamic property reads where the property name is imprecise, without the use of syntactic patterns.

² In JavaScript, reading an absent property yields the special value `undefined`.

³ `AnyString` is an abstract value that represents any string. In practice we instead use a slightly more precise abstract value representing `AnyString\{"map", "trim"}`.

Free variable partitioning

A second instantiation of value partitioning is for handling free variables more precisely. In the example, this is useful for `func` in line 9, which is a free variable in the function defined in lines 7–10. At that function definition, we partition both the resulting abstract function value ℓ and the abstract value of `func` according to the existing partitioning of `func`, intuitively to be able to distinguish functions created with different values of the free variable. This means that the function value being written at the dynamic property write in line 7 is $[t_1 \mapsto \ell_{t'_1}, t_2 \mapsto \ell_{t'_2}, t_3 \mapsto \ell_{t'_3}]$ where $\ell_{t'_1}$ represents the function created at a point where `func` is `f1` (i.e., that point is at the end of a t_1 trace), and similarly for the other partitions. At the same time, the value of `func` becomes $[t_1 \mapsto \mathbf{f1}, t_2 \mapsto \mathbf{f2}, t_3 \mapsto \mathbf{undefined}, t'_1 \mapsto \mathbf{f1}, t'_2 \mapsto \mathbf{f2}, t'_3 \mapsto \mathbf{undefined}]$ where the three new partitions t'_1 , t'_2 , and t'_3 denote the new partitioning we have made (one abstract value can thus have multiple partitionings simultaneously). Using the property name partitioning mechanism described above, at the dynamic property write in line 7, $\ell_{t'_1}$ is written to the `map` property of `object.prototype`, and similarly for the other properties.

We can exploit the free variable partitioning information when the function is later called. Assume the analysis encounters a call to the `map` method. The abstract value of `lodash.prototype.map` is then $\ell_{t'_1}$. We now use t'_1 as a context in ordinary context sensitive analysis of the function, so that when reaching `func` in line 9, it suffices to consider only the t'_1 partition of `func`, which yields the precise value `f1`, so again, we successfully avoided mixing together the properties.

Type partitioning

The above two uses of value partitioning are sufficient for analyzing the motivating example without critical precision losses, but we can make the analysis even more precise using a third variant. The function named `isFunction` used in the branch condition in line 6 is a typical example of a *predicate function*, i.e., a one-parameter function that returns a boolean, in this case testing whether the value passed in is a function. Assume the abstract value of the argument `object` is `fun1|obj2`, meaning that it represents either a function `fun1` or a non-function object `obj2`. With a simple analysis, the abstract return value and hence the branch condition is `Bool` representing any boolean value, so the analysis does not know that `object` cannot be `obj2` inside the branch. This causes the analysis to spuriously raise a type error when writing to `object.prototype` in line 7.

Type partitioning avoids that imprecision as follows. Type partitioning is triggered at any call to a function with one argument, and partitions that argument according to its types. In this case, the value of `object` is partitioned into $[a \mapsto \mathbf{fun1}, b \mapsto \mathbf{obj2}]$. The result value from `isFunction` then becomes $[a \mapsto \mathbf{true}, b \mapsto \mathbf{false}]$, which we can exploit using ordinary control sensitivity [10] (also called type refinement [14]) at the “true” branch such that `object` in line 7 will only be `fun1` and not `obj2`.

Overview

In Section 3 we give a brief introduction to the analysis domain of TAJ. Section 4 explains the general value partitioning mechanism, and Section 5 details the three instantiations: property name partitioning, free variable partitioning, and type partitioning. Section 6 describes our experimental evaluation, and Section 7 discusses related work.

$r_1[r_2] \leftarrow r_3:$	Writes r_3 to the property named r_2 of the object r_1
$r_1 \leftarrow r_2[r_3]:$	Reads the property named r_3 of the object r_2 to r_1
$r_1 \leftarrow x:$	Reads the value of the variable x to r_1
$x \leftarrow r_1:$	Writes r_1 to the variable x
$r_1 \leftarrow c:$	Assigns the constant c to r_1
$r_1 \leftarrow \text{function}(x)\{\dots\}:$	Creates a closure for the function and stores it in r_1
$\text{if}(r_1):$	Conditionally propagates dataflow (to model <code>if</code> and <code>while</code>)
$r_1 \leftarrow r_2(r_3):$	Calls the function r_2 with argument r_3 and stores the result in r_1
$r_1 \leftarrow r_2 \oplus r_3:$	Computes the binary operation $r_2 \oplus r_3$ and stores the result in r_1

■ **Figure 2** The main flow graph instructions in TAJJS.

$n \in N$: Nodes	$X \in \text{AnalysisLattice} = L \rightarrow \text{State}$
$c \in C$: Contexts	$\sigma \in \text{State} = (L \rightarrow \text{Obj}) \times \text{Registers}$
$p \in P$: Property names	$o \in \text{Obj} = P \rightarrow \text{Value}$
$\ell \in L = N \times C$: Locations	$r \in \text{Registers} = R \rightarrow \text{Value}$
	$v \in \text{Value} = \text{Prim} \times \mathcal{P}(L)$

■ **Figure 3** Simplified abstract domain.

3 Background: The TAJJS Analyzer

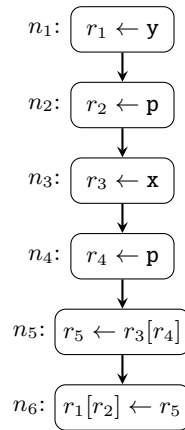
In this section we give a brief introduction to a heavily simplified version of the analysis domain and program representation used in TAJJS [11, 2], which lays the foundation for our extensions in the following sections.

TAJJS is an open-source dataflow analysis tool for JavaScript built as a monotone framework [12]. A JavaScript program is represented as a control flow graph for each function, with nodes representing primitive instructions of the different kinds listed in Figure 2. Each instruction operates on registers, which can be thought of as special local variables. For simplicity, we ignore `this` and receiver objects at calls, and we assume all functions have only one parameter. As an example, the single JavaScript statement $\mathbf{y}[\mathbf{p}] = \mathbf{x}[\mathbf{p}]$ is represented as six flow graph nodes as shown in Figure 4.

The components of the abstract domain are summarized in Figure 3. A *location* is a pair of a node and a context. The contexts allow for context sensitivity (using the context-sensitivity strategy described by Andreasen and Møller [2]). The main abstract domain, *AnalysisLattice*, is a lattice that maps locations to abstract states, where each state contains abstract values of object properties and registers. Objects are modeled using context-sensitive allocation-site abstraction [6, 20], so abstract object addresses are simply locations.⁴ Functions are special kinds of objects. Abstract values are modeled using a product of a constant-propagation lattice [15] named *Prim* of primitive values (strings, numbers, etc.) and a powerset lattice of object addresses.

The analysis is control sensitive by pruning infeasible dataflow at `if` nodes. This includes not only eliminating flow along unreachable branches, for example when a branch condition is definitely false [27], but also filtering abstract values based on the branch condition [10, 14].

⁴ TAJJS models absence/presence of object properties and uses two artificial properties `DEFAULTNUMERIC` and `DEFAULTOTHER` to model properties with unknown numeric/non-numeric names; we ignore that here.



■ **Figure 4** Fragment of a control flow graph, for the single statement $y[p] = x[p]$.

As an example, the JavaScript code `if(z)` is represented by two primitive instructions, $r_6 \leftarrow z$ and `if(r_6)`. In the “true” branch, not only r_6 but also z must have the value `true`.⁵ To track the connection between r_6 and z , a simple intraprocedural must-equals analysis is performed alongside the main dataflow analysis. We leverage this mechanism in Section 5, for example to obtain the information that r_2 , r_4 , and p must have the same value at the property read operation in Figure 4 (unless a property accessor changes p). To keep Figure 3 simple, we omit the must-equals information in the description of the *State* lattice.

In the following sections, with a slight abuse of notation we let $\sigma(r)$ denote the value of register r in state σ , and similarly, $\sigma(\mathbf{x})$ denotes the value of variable \mathbf{x} . Also, we use the notation $\sigma(r) := \dots$ to describe the operation of writing a given value to register r and also to the variables and registers that are equal to r according to the must-equals information. If $\ell \in L$ is a location representing an object address, we sometimes write ℓ for the abstract value $(\perp, \{\ell\}) \in \text{Value}$. Similarly, for abstract values that represent primitive values only, we omit the location sets, for example, “foo” denotes the abstract value $(\text{“foo”}, \emptyset) \in \text{Value}$.

We omit many details of TAJIS, including the definitions of the concretizations of the lattice elements, the definitions of the transfer functions for the different instructions, how values of variables are being stored in special activation objects, and how a call graph is built during the analysis. Analyzing full JavaScript also requires reasoning about prototypes, scope chains, implicit type conversions, exceptions, the standard library, property accessors (getters and setters), and much more. It suffices to know that the resulting abstract states soundly overapproximate the possible program behavior [7].

A *trace* is a concrete execution of the program expressed as a finite sequence of pairs (ℓ, γ) where ℓ is a location and γ is a concrete state, starting at the program entry point with the initial call context in an empty state. The semantics of a program is defined as a set of traces. The *collecting semantics* is the program semantics projected onto the program locations: Given a location ℓ , the collecting semantics for ℓ , denoted $\llbracket \ell \rrbracket$, is the set of states that appear at ℓ in the set of traces defined by the program semantics. The analysis result is thus a lattice element $X \in \text{AnalysisLattice}$ such that $\llbracket \ell \rrbracket$ is a subset of the concretization of $X(\ell)$ for all locations $\ell \in L$.

⁵ In actual JavaScript, the value must be *truthy*, which also includes nonempty strings, nonzero numbers, and objects.

$$\begin{aligned}
t \in T & : \text{Partition tokens} \\
o \in \text{Obj} & = P \rightarrow \text{PartitionedValue} \\
r \in \text{Registers} & = R \rightarrow \text{PartitionedValue} \\
pv \in \text{PartitionedValue} & = T \leftrightarrow \text{Value}
\end{aligned}$$

■ **Figure 5** Extension of the abstract domain for value partitioning.

4 Value Partitioning

To prepare the analysis for value partitioning, we introduce a set T of partition tokens and replace occurrences of *Value* by *PartitionedValue* in the abstract domain, as shown in Figure 5. A *partitioned value* is a partial map from partition tokens to ordinary values. We use the notation $[t_1 \mapsto v_1, \dots, t_k \mapsto v_k]$ (or set-builder notation like $[t_i \mapsto v_i \mid i = 1, \dots, k]$) to denote the partitioned value that maps t_i to v_i for each $i = 1, \dots, k$ and is undefined for all other partition tokens.

The partition tokens play a similar role as in trace partitioning [23], but at the level of abstract values. (We explain the differences between value partitioning and traditional trace partitioning in more detail in Section 7.) A partition token intuitively represents a set of execution traces. The special token ANY represents all traces, so the partitioned value $[\text{ANY} \mapsto v]$ has the same meaning as the ordinary value v in the original abstract domain. As an invariant, all partitioned values we use are defined for the token ANY.⁶ We extend partitioned values to be total functions $pv: T \rightarrow \text{Value}$ by defining $pv(t) = pv(\text{ANY})$ when $t \notin \text{dom}(pv)$.⁷

Assume $X \in \text{AnalysisLattice}$ is the result of analyzing a given program, $\sigma = X(\ell)$ is the abstract state at some location ℓ , and $[\dots, t \mapsto v, \dots] = \sigma(r)$ is the partitioned value of some register r . The meaning of such a partitioned value is that for any trace that ends at ℓ and is in the set of traces represented by t , the concrete value of r is in the concretization of the abstract value v .

A *covering*⁸ at a location ℓ is a set of partition tokens where the union of the sets of traces they represent is the set of all traces that lead to ℓ . This means that if $\sigma(\mathbf{x}) = [\dots, t_1 \mapsto v_1, \dots, t_k \mapsto v_k, \dots]$ where $\sigma = X(\ell)$ for some program variable \mathbf{x} at location ℓ where $\{t_1, \dots, t_k\}$ is a covering, then for every concrete state in $\llbracket \ell \rrbracket$, the value of \mathbf{x} is in the concretization of at least one of the abstract values v_1, \dots, v_k . For the initial abstract state at the program entry, all partitioned values use the trivial covering $\{\text{ANY}\}$.

Now that we have generalized the abstract domain, it is easy to adjust all transfer functions for the different kinds of nodes to operate on partitioned values instead of ordinary values. As an example, the original transfer function for $r_1 \leftarrow r_2 \oplus r_3$ updates a given abstract state σ by $\sigma(r_1) := \sigma(r_2) \oplus \sigma(r_3)$ (where \oplus applied to abstract values works as in constant propagation).⁹

⁶ When we define a partitioned value $[t_i \mapsto v_i \mid i = 1, \dots, k]$ without an ANY token, an ANY partition is implicitly created with value $v_1 \sqcup \dots \sqcup v_k$.

⁷ In trace partitioning terminology, this use of ANY corresponds to a simple pre-ordering of partition tokens.

⁸ For formal definitions of the notions of traces and coverings, see Rival and Mauborgne [23]. Basing our approach on partitions instead of coverings (a *partition* is a covering where all the trace sets are disjoint) could improve precision but would complicate the analysis without much practical benefit.

⁹ The actual TAJs analysis also models implicit type conversions.

16:10 Value Partitioning

T	:: =	ANY	(Section 4)
		VAL $\langle N, R, Value \rangle$	(Section 5.1)
		FUN $\langle F, C, T \rangle$	(Section 5.2)
		TYPE $\langle N, R, Types \rangle$	(Section 5.3)
$Types$:: =	undefined null number string boolean	
		object array function regexp	

■ **Figure 6** Partition tokens used by property name partitioning, free variable partitioning, and type partitioning.

When switching to the domain with partitioned values, we simply replace $\sigma(r_2) \oplus \sigma(r_3)$ by $[t \mapsto pv_2(t) \oplus pv_3(t) \mid t \in \text{dom}(pv_2) \cup \text{dom}(pv_3)]$ where $pv_2 = \sigma(r_2)$ and $pv_3 = \sigma(r_3)$. The other transfer functions and least-upper-bound are adapted similarly.

A small example can illustrate how partitioning can make the analysis relational. Assume the binary operation is equality, $r_1 \leftarrow r_2 == r_3$, and that we have two partitions, t_1 and t_2 , where both registers r_2 and r_3 have the value 42 in partition t_1 , and both have the value "foo" in partition t_2 . With partitioning, the value of r_1 becomes $[t_1 \mapsto \mathbf{true}, t_2 \mapsto \mathbf{true}]$ (i.e., definitely true), whereas without partitioning, r_2 and r_3 both have the value 42|"foo", so the value of r_1 becomes **AnyBool** (i.e., true or false).

To get any advantage of the new abstract domain, we of course need to modify specific transfer functions to selectively introduce partition tokens and further exploit the extra information available regarding relational properties between values. We show how that can be accomplished in Section 5. Those mechanisms rely on some general operations for manipulating the partitions in partitioned values. Most importantly, we use an operation \uplus when introducing new coverings: $pv_1 \uplus pv_2$ where $pv_1, pv_2 \in \text{PartitionedValue}$ denotes the combined partitioned value. For each token that is only present in one of pv_1 or pv_2 , the new value will be the value for that token, and for each token shared by pv_1 and pv_2 , the new value will be the join of the two respective values.

5 Three Instantiations of Value Partitioning

We now present three instantiations of the value partitioning framework. Each of them targets a category of relational properties that are relevant to analysis of JavaScript libraries. Each instantiation introduces a family of partition tokens, as shown in Figure 6, along with some modification of the analysis transfer functions. Each partition token represents a set of traces, as explained in the following.

5.1 Property Name Partitioning

The first use of value partitioning is for improving precision at correlated object property read/write operations as in the motivating example.

Partition tokens for property name partitioning

We introduce a family of partition tokens, $\text{VAL}\langle n, r, v \rangle$, where $n \in N$, $r \in R$, and $v \in \text{Value}$. Such a token represents the set of traces where at the last occurrence of n , the value of register r is v . In all $\text{VAL}\langle n, r, v \rangle$ tokens we use in property name partitioning, the node n

$$\sigma(r_3) := \begin{cases} \sigma(r_3) \uplus [\text{VAL}\langle n, r_3, p \rangle \mapsto p \mid p \in \text{PROPNames}(\sigma(r_2))] & \text{if } \sigma(r_3)(\text{ANY}) = \text{AnyString} \\ \uplus [\text{VAL}\langle n, r_3, \text{OTHER} \rangle \mapsto \text{AnyString}] & \text{otherwise} \\ \sigma(r_3) & \end{cases}$$

$$\sigma(r_1) := \begin{cases} [\text{VAL}\langle n, r_3, p \rangle \mapsto \text{READPROP}(\sigma(r_2)(\text{ANY}), p) \mid p \in \text{PROPNames}(\sigma(r_2))] & \text{if } \sigma(r_3)(\text{ANY}) = \text{AnyString} \\ \uplus [\text{VAL}\langle n, r_3, \text{OTHER} \rangle \mapsto \text{undefined}] & \text{otherwise} \\ [\text{ANY} \mapsto \text{READPROP}(\sigma(r_2)(\text{ANY}), \sigma(r_3)(\text{ANY}))] & \end{cases}$$

■ **Figure 7** Introduction of partitioned values at a dynamic property read node n of the form $r_1 \leftarrow r_2[r_3]$.

is a property read node (i.e., of the form $r_1 \leftarrow r_2[r_3]$), the register r is the one holding the property name in that instruction (i.e., r_3 in $r_1 \leftarrow r_2[r_3]$), and the value v is a property name (i.e., an element of P).¹⁰

As an example, assume the code from Figure 4 appears inside a loop, and consider the following two traces that both end at n_6 :

$$\tau_a = \dots (n_1, \gamma_{1a})(n_2, \gamma_{2a})(n_3, \gamma_{3a})(n_4, \gamma_{4a})(\mathbf{n_5}, \gamma_{5a})(n_6, \gamma_{6a})$$

and

$$\tau_b = \dots (n_1, \gamma_{1b})(n_2, \gamma_{2b})(n_3, \gamma_{3b})(n_4, \gamma_{4b})(\mathbf{n_5}, \gamma_{5b})(n_6, \gamma_{6b})$$

where each “ \dots ” is a trace prefix leading from the program entry point to this part of the code, $\gamma_{1a}, \dots, \gamma_{6b}$ are concrete states, and τ_a is a prefix of τ_b . The last occurrence of n_5 (which is the instruction $r_5 \leftarrow r_3[r_4]$) is emphasized in each of the traces. Also assume that the value of the register r_4 is “foo” in γ_{5a} and “bar” in γ_{5b} . Note that r_4 is the register holding the property name at the n_5 instruction. In this situation, the token $\text{VAL}\langle n_5, r_4, \text{“foo”} \rangle$ represents τ_a but not τ_b .

Dynamic property reads

Figure 7 shows the modified transfer function for read-property nodes, $r_1 \leftarrow r_2[r_3]$. The function $\text{READPROP}(v_1, v_2)$ looks up the abstract value of properties named v_2 in the abstract objects pointed to by v_1 in the current state σ .¹¹ Property name partitioning is triggered if the property name is not precise (here modeled as `AnyString`), so in that case we partition the property name r_3 with respect to the properties that appear in the abstract objects pointed to by r_2 (expressed as $\text{PROPNames}(\sigma(r_2))$), and perform the property read for each partition to obtain a partitioned value for r_1 . We use the artificial abstract value `OTHER` \in *Value* to represent all other properties; for that partition, the result value becomes `undefined`.¹² If

¹⁰ In JavaScript, property names are either strings, which we model in the sub-lattice *Prim*, or symbols, which can be modeled as special heap locations.

¹¹ Reading an object property is a nontrivial operation in JavaScript because of prototypes, getters, and implicit type conversions. Importantly, the value partitioning mechanism is orthogonal to such JavaScript technicalities.

¹² In our implementation we use a more precise string lattice, which allows us to express more precisely that $\sigma(r_3)$ for the $\text{VAL}\langle n, r_3, \text{OTHER} \rangle$ partition is $\text{AnyString} \setminus \text{PROPNames}(\sigma(r_2))$, i.e., any string except for the property names that are covered by other partitions. See also footnote 3.

16:12 Value Partitioning

for each $t \in \text{CHOOSECOMMONCOVERING}(\sigma(r_2), \sigma(r_3))$:
 $\text{WRITEPROP}(\sigma(r_1)(\text{ANY}), \sigma(r_2)(t), \sigma(r_3)(t))$

■ **Figure 8** Exploiting partitioned values at a dynamic property write node, $r_1[r_2] \leftarrow r_3$.

the property name r_3 is already precise (corresponding to the “otherwise” cases), there is no need to introduce new partitions, so in that case r_3 is unmodified and the result value r_1 is obtained directly using `READPROP` and the `ANY` partition token.

Recall that a $\text{VAL}\langle n, r, p \rangle$ token represents the set of traces where at the last occurrence of n , the value of register r is v . To respect this property we need to remove all existing $\text{VAL}\langle n, _, _ \rangle$ tokens from the abstract state before applying the transfer function for dynamic property reads. (This is safe because every abstract value still has other coverings, in particular `{ANY}`.)

Notice that for both r_3 and r_1 , the new partitions use the partition tokens $\text{VAL}\langle n, r_3, p \rangle$ where n is the read-property node. Evidently, the new partition tokens form a covering. Also, this new transfer function respects the interpretation of the newly added $\text{VAL}\langle n, r, p \rangle$ tokens, and due to the partitioning, the resulting abstract states maintain the relation between the involved object property names and values.

Dynamic property writes

Next, we modify the transfer function for dynamic property writes, $r_1[r_2] \leftarrow r_3$, as shown in Figure 8, to take advantage of the partitionings introduced at dynamic property reads. The function $\text{WRITEPROP}(v_1, v_2, v_3)$ writes v_3 to the properties named v_2 in the objects referred to by v_1 .¹³ The function $\text{CHOOSECOMMONCOVERING}$ finds a covering shared by the property name $\sigma(r_2)$ and the value to be written $\sigma(r_3)$. (An example is given below.) If multiple such coverings exist, a largest one (i.e., one with the largest number of partition tokens) is selected.¹⁴ Recall that the two values always share the `{ANY}` covering, which will be used if no other covering exist. When a covering has been chosen, the value is written to the appropriate object property for each partition, thereby exploiting the relational information. In case the `{ANY}` covering is chosen, the transfer function behaves as the original version without value partitioning.

Example

To better understand property name partitioning, we now explain the mechanism in more detail on the example given in Figure 4. Let us assume that $\sigma(\mathbf{p}) = [\text{ANY} \mapsto \text{AnyString}]$, $\sigma(\mathbf{x}) = [\text{ANY} \mapsto \text{obj}_2]$ and $\sigma(\mathbf{y}) = [\text{ANY} \mapsto \text{obj}_1]$ where in state σ , obj_1 is the location of an empty abstract object and obj_2 is the location of an abstract object with two properties, `{foo: 1, bar: 2}`. This means when analyzing the read property node $r_5 \leftarrow r_3[r_4]$ we have $\sigma(r_3) = [\text{ANY} \mapsto \text{obj}_2]$ and $\sigma(r_4) = [\text{ANY} \mapsto \text{AnyString}]$. Since the property name r_4 is

¹³We omit the details of how the implementation of `WRITEPROP` in TAJE handles strong/weak updates, setters, and implicit type conversions. Importantly, the value partitioning mechanism is orthogonal to such JavaScript technicalities.

¹⁴Multiple coverings can arise if, for example, the same property name is used at two different property read operations. We choose the largest covering based on the heuristic that fine-grained coverings lead to higher precision than coarse-grained coverings. The most important consequence of this heuristic is that we avoid the `{ANY}` covering if others are available. In case of multiple largest ones, an arbitrary one is selected among them.

$$\begin{aligned}
pv \in \text{PartitionedValue} &= T \leftrightarrow \text{FPValue} \\
fv \in \text{FPValue} &= \text{FunctionPartitions} \times \text{Value} \\
fp \in \text{FunctionPartitions} &= \mathcal{P}(T) \\
\text{AnalysisLattice} &= C' \times N \rightarrow \text{State} \\
c \in C' &= \text{FunctionPartitions} \times C
\end{aligned}$$

■ **Figure 9** Extensions of the abstract domain for free variable partitioning.

imprecise, the first case in each definition in Figure 7 applies, meaning that value partitioning is triggered. Since $\text{PROPNames}(\sigma(r_2)) = \{\text{"foo"}, \text{"bar"}\}$, we update r_4 such that $\sigma(r_4)$ equals $[\text{VAL}\langle n, r_4, \text{"foo"} \rangle \mapsto \text{"foo"}, \text{VAL}\langle n, r_4, \text{"bar"} \rangle \mapsto \text{"bar"}, \text{VAL}\langle n, r_4, \text{OTHER} \rangle \mapsto \text{AnyString}]$, where n is the read property node. Recall from Section 3 that the operation $\sigma(r_4) := \dots$ does not only modify r_4 but also the must-equals variables and registers, meaning that this partitioned value is also written to r_2 and \mathbf{p} . The value being read gets the same partitions, such that $\sigma(r_5)$ becomes $[\text{VAL}\langle n, r_4, \text{"foo"} \rangle \mapsto 1, \text{VAL}\langle n, r_4, \text{"bar"} \rangle \mapsto 2, \text{VAL}\langle n, r_4, \text{OTHER} \rangle \mapsto \text{undefined}]$.

When reaching the property write operation $r_1[r_2] \leftarrow r_5$, the state σ contains $\sigma(r_2) = [\text{VAL}\langle n, r_4, \text{"foo"} \rangle \mapsto \text{"foo"}, \text{VAL}\langle n, r_4, \text{"bar"} \rangle \mapsto \text{"bar"}, \text{VAL}\langle n, r_4, \text{OTHER} \rangle \mapsto \text{AnyString}]$ and $\sigma(r_5) = [\text{VAL}\langle n, r_4, \text{"foo"} \rangle \mapsto 1, \text{VAL}\langle n, r_4, \text{"bar"} \rangle \mapsto 2, \text{VAL}\langle n, r_4, \text{OTHER} \rangle \mapsto \text{undefined}]$. We now apply the transfer function from Figure 8. The two values $\sigma(r_2)$ and $\sigma(r_5)$ share two coverings: $\{\text{ANY}\}$ and $\{\text{VAL}\langle n, r_4, \text{"foo"} \rangle, \text{VAL}\langle n, r_4, \text{"bar"} \rangle, \text{VAL}\langle n, r_4, \text{OTHER} \rangle\}$. Since the second covering is largest, that one is picked by $\text{CHOOSECOMMONCOVERING}(\sigma(r_2), \sigma(r_5))$. We therefore perform three writes corresponding to the abstract assignments $\text{obj}_1[\text{"foo"}]=1$, $\text{obj}_1[\text{"bar"}]=2$, and $\text{obj}_1[\text{AnyString}]=\text{undefined}$; notably, the properties **foo** and **bar** are not mixed together.

5.2 Free Variable Partitioning

We now explain how to leverage value partitioning to gain precision for free variables, such as **func** in line 9 in the motivating example from Figure 1.

Extending the abstract domain

The first step is to extend the abstract domain as shown in Figure 9. The *Value* component in *PartitionedValue* is replaced by *FPValue*, which is a product of *FunctionPartitions* and *Value*. The component *FunctionPartitions* is a set of partition tokens, which we use for tracking which partitions the functions described in the *Value* component may have been declared in. (For instance, for the motivating example from Figure 1, the function declared in lines 7–10 was created in the partitions t'_1, t'_2 , and t'_3 so the corresponding abstract values become¹⁵ $(\{t'_1\}, \ell)$, $(\{t'_2\}, \ell)$, and $(\{t'_3\}, \ell)$, where ℓ denotes the location for the created closure.) To preserve this information when analyzing calls to such functions, we also augment the set of contexts to include this information (replacing C by C' in *AnalysisLattice*). The *FunctionPartitions* set is empty for values and contexts that do not use free variable partitioning.

¹⁵These three abstract values are denoted $\ell_{t'_1}$, $\ell_{t'_2}$, and $\ell_{t'_3}$, respectively, in the motivating example in Section 2.

$$pv(t) = \begin{cases} pv(t) & \text{if } t \in \text{DOM}(pv) \\ \perp & \text{otherwise if } t = \text{FUN}\langle f, c, t' \rangle \wedge \\ & \exists c', t'' : c \neq c' \wedge \text{FUN}\langle f, c', t'' \rangle \in \text{DOM}(pv) \\ pv(\text{ANY}) & \text{otherwise} \end{cases}$$

■ **Figure 10** Redefining how partitioned values are extended to total functions, exploiting free variable partitioning.

Next, we introduce a new kind of partition tokens, and we then describe how elements of *FunctionPartitions* are created at function expressions and used at read variable nodes.

Partition tokens for free variable partitioning

We introduce a new kind of partition tokens, $\text{FUN}\langle f, c, t \rangle$, where f is a function, $c \in C'$ is a context, and $t \in T$ is a partition token. A trace is represented by such a token if (1) the trace ends at a program location that belongs to a closure that was created when the trace up to that point was a t trace, and (2) that point in the trace is in function f in context c . (For instance, in the motivating example, a trace ending in line 9 where the currently executed closure was created in line 7 at the end of a t_1 trace can be represented by $\text{FUN}\langle f, c, t_1 \rangle$, where f is the function at lines 2–11 and c is the context for the call to that function.) We only allow such partition tokens to appear in abstract values of variables that are declared in f . Intuitively, we use these partition tokens to obtain a form of heap specialization (also called heap cloning or context sensitive heap) [20] for the activation objects of f .¹⁶

An important property is that if the abstract value of a variable \mathbf{x} declared in a function f contains partition token $\text{FUN}\langle f, c', t'' \rangle$ for some c', t'' but not $\text{FUN}\langle f, c, t' \rangle$ for any c, t' where $c \neq c'$, then f has not been invoked with context c in any trace represented by $\text{FUN}\langle f, c, t' \rangle$. This means that it is safe to redefine how partitioned values are extended to total functions as shown in Figure 10. The only difference between the new and the original definition from Section 4 is the second case, where \perp is returned to indicate that the set of traces for the given partition is empty due to the above mentioned property being satisfied.

Function definitions

Assume the analysis reaches a function definition node, $r_1 \leftarrow \text{function}(\dots)\{\dots\}$, while analyzing a function f in context c , and that the function being defined has free variables $\mathbf{x}_1, \dots, \mathbf{x}_n$ that are declared in f (i.e., as parameters or local variables). Note that f is the function containing the function definition node being analyzed, not the function being defined. Let ℓ denote the location of the newly created closure according to the original transfer function without free variable partitioning. We now partition both the resulting function value of register r_1 and the values of $\mathbf{x}_1, \dots, \mathbf{x}_n$ as shown in Figure 11.

First, we use a function `CHOOSECOVERING` that finds a largest covering, denoted LC , among the values of $\mathbf{x}_1, \dots, \mathbf{x}_n$. (If multiple such coverings exist, an arbitrary one is selected among them, as before.)

¹⁶ Local variables and arguments are stored as properties on activation objects, which are created on each invocation.

$$\begin{aligned}
LC &= \text{CHOOSECOVERING}(\sigma(\mathbf{x}_1), \dots, \sigma(\mathbf{x}_n)) \\
\sigma(\mathbf{x}_i) &:= \begin{cases} \sigma(\mathbf{x}_i) \uplus [\text{FUN}\langle f, c, t \rangle \mapsto \sigma(\mathbf{x}_i)(t) \mid t \in LC] & \text{if } LC \subseteq \text{dom}(\sigma(\mathbf{x}_i)) \\ \sigma(\mathbf{x}_i) & \text{otherwise} \end{cases} \\
\sigma(r_1) &:= \begin{cases} [t \mapsto (\{\text{FUN}\langle f, c, t \rangle\} \cup fp, \ell) \mid t \in LC] & \text{if } LC \subseteq \text{dom}(\sigma(\mathbf{x}_i)) \text{ for some } \mathbf{x}_i \\ [\text{ANY} \mapsto (\emptyset, \ell)] & \text{otherwise} \end{cases}
\end{aligned}$$

■ **Figure 11** Introduction of partitioned values at a function definition node $r_1 \leftarrow \text{function}(\dots)\{\dots\}$.

For each \mathbf{x}_i for $i = 1, \dots, n$, if the current value of \mathbf{x}_i contains the covering LC , we add $\text{FUN}\langle f, c, t \rangle \mapsto \sigma(\mathbf{x}_i)(t)$ to the value of \mathbf{x}_i for each $t \in LC$. (This evidently respects the meaning of $\text{FUN}\langle f, c, t \rangle$ tokens informally described in the beginning of the section.) Otherwise, if the current value of \mathbf{x}_i does not contain LC , we leave \mathbf{x}_i unmodified.

For the result register r_1 , we augment the function location ℓ by the same partition tokens. If at least one free variable has been partitioned (i.e., $LC \subseteq \text{dom}(\sigma(\mathbf{x}_i))$ for some \mathbf{x}_i), then for each of the partition tokens $t \in LC$, the value of r_1 becomes the augmented value $(\{\text{FUN}\langle f, c, t \rangle\} \cup fp, \ell)$ where fp is the set of function partitions in the current context c . By augmenting the value using the $\text{FUN}\langle f, c, t \rangle$ token, the information about the partitioning is available when ℓ is later invoked, which is explained below. (The function partitions fp of the current context describe how the current function was declared in an outer scope, so by inheriting those, the partitioning also works for multiple layers of nested functions.) Otherwise, if none of the free variables have been partitioned, register r_1 is assigned the partitioned value $[\text{ANY} \mapsto (\emptyset, \ell)]$, which is equivalent to the original transfer function without free variable partitioning.

Function calls

At a function call $r_1 \leftarrow r_2(r_3)$ where $\sigma(r_2)$ is an augmented function value (fp, v) (i.e., fp is a set of partition tokens introduced at function definitions and v refers to the set of closures that may be invoked), we use fp to augment the context for each callee. (The set of augmented contexts C' contains the *FunctionPartitions* component exactly for this purpose.) Assume for simplicity that v refers to a single closure location so we only have one callee. By augmenting the context, when analyzing the body of the callee we retain the information about the partitions where the callee closure was created, which we can exploit when reading its free variables as explained next.

Variable reads

Figure 12 shows the updated transfer function for read variable nodes $r_1 \leftarrow \mathbf{x}$, where we read a variable \mathbf{x} in a calling context with function partitions fp . The set of function partitions fp tells us which partitions the current closure may have been created in. For this reason, if the abstract value of \mathbf{x} contains partition tokens that are also in fp , we can obtain a covering for \mathbf{x} by considering only those partition tokens. If there is no such partition token, we just read the value of \mathbf{x} as in the original transfer function.

16:16 Value Partitioning

$$\sigma(r_1) := \begin{cases} [\text{ANY} \mapsto \bigsqcup\{\sigma(\mathbf{x})(t) \mid t \in \text{dom}(\sigma(\mathbf{x})) \cap fp\}] & \text{if } \text{dom}(\sigma(\mathbf{x})) \cap fp \neq \emptyset \\ \sigma(\mathbf{x}) & \text{otherwise} \end{cases}$$

■ **Figure 12** Exploiting partitioned values at a read variable node, $r_1 \leftarrow \mathbf{x}$.

```
22 function f(v) {
23   return function g() {
24     return v;
25   }
26 }
27
28 var foo = f("foo");
29 var bar = f("bar");
30
31 assert(bar() != "foo");
```

■ **Figure 13** Free variable partitioning example with different contexts.

```
32 var o1 = {x: 1, y: 2};
33 var o2 = {};
34 Object.keys(o1).forEach(
35   function h(p) {
36     var v = o1[p];
37     o2[p] = function j() {
38       return v;
39     }
40   }
41 );
42 assert(o2.y() != 1);
```

■ **Figure 14** Free variable partitioning example with partitioned argument.

As an example, assume $\sigma(\mathbf{x}) = [\text{FUN}\langle f, c, t \rangle \mapsto 1, \text{FUN}\langle f, c', t' \rangle \mapsto 2, \dots]$ and $fp = \{\text{FUN}\langle f, c, t \rangle\}$. The value of \mathbf{x} tells us that \mathbf{x} must be a local variable in function f which may have been called in contexts c and context c' , and that \mathbf{x} 's value is 1 or 2, respectively. Since $fp = \{\text{FUN}\langle f, c, t \rangle\}$, we know that the current function is defined inside the lexical scope of f in context c , meaning that the value of \mathbf{x} must be 1.

Examples

To better understand free variable partitioning, we provide two examples. The first example (Figure 13) shows how free variable partitioning can preserve precision when a function is called in multiple contexts, in a way that resembles traditional heap specialization [20]. The second example (Figure 14) shows how free variable partitioning can preserve the precision of free variables partitioned with property name partitioning.

In Figure 13, lines 22–26 define a function \mathbf{f} that returns a closure, which on invocation returns the argument passed to \mathbf{f} . Lines 28 and 29 call \mathbf{f} with the arguments "foo" and "bar" and store the returned closures in the variables \mathbf{foo} and \mathbf{bar} , respectively. Line 31 calls the closure stored in \mathbf{bar} and asserts that the resulting value is not the string "foo". The two calls to \mathbf{f} are analyzed in different contexts c and c' (due to the context sensitivity mechanism mentioned in Section 3, as "foo" and "bar" are determinate values). For the invocation $\mathbf{bar}()$, the resulting value is the value of the free variable \mathbf{v} in the closure stored in \mathbf{bar} . If not using heap specialization, the two concrete activation objects at the two calls to \mathbf{f} would be modeled by a single abstract object, so the free variable \mathbf{v} would have the imprecise abstract value `AnyString`. To reason precisely about the assertion in line 31, the analysis has to distinguish the value of \mathbf{v} at the two calls. The baseline TAJIS analyzer accomplishes this by the use of heap specialization [2], which provides two different abstract activation objects for the calls to \mathbf{f} , so the two values "foo" and "bar" are kept separate.

With free variable partitioning we obtain the same degree of precision as with heap specialization in this situation. Since v is a free variable in the closure created in line 23, we apply the top cases in the transfer functions shown in Figure 11 with $LC = \{\text{ANY}\}$. This means that v after the call to $\mathbf{f}(\text{"foo"})$ will have the value $[\text{ANY} \mapsto \text{"foo"}, \text{FUN}\langle f, c, \text{ANY} \rangle \mapsto \text{"foo"}]$ and the value written to the `foo` variable is $(\{\text{FUN}\langle f, c, \text{ANY} \rangle\}, \ell_g)$ where ℓ_g is the location of the closure created in line 23. For the call to $\mathbf{f}(\text{"bar"})$, the value for v will similarly be $[\text{ANY} \mapsto \text{"bar"}, \text{FUN}\langle f, c', \text{ANY} \rangle \mapsto \text{"bar"}]$ and the value written to `bar` is $(\{\text{FUN}\langle f, c', \text{ANY} \rangle\}, \ell_g)$. Note the difference in the context part of the FUN token (c at the "foo" call and c' at the "bar" call), since the calls to \mathbf{f} are in those two different contexts. The value of v then becomes $[\text{ANY} \mapsto \text{AnyString}, \text{FUN}\langle f, c, \text{ANY} \rangle \mapsto \text{"foo"}, \text{FUN}\langle f, c', \text{ANY} \rangle \mapsto \text{"bar"}]$, so that the FUN partitions preserve the precise values.

Now when analyzing `bar()`, `bar` has the value $(\{\text{FUN}\langle f, c', \text{ANY} \rangle\}, \ell_g)$, which means the calling context to the function \mathbf{g} is augmented with the set of function partitions $\{\text{FUN}\langle f, c', \text{ANY} \rangle\}$ as described above. When reading the free variable v in line 24, we use the first case in the transfer function defined in Figure 12, since $\text{dom}(\sigma(\mathbf{x})) \cap fp$ is $\{\text{FUN}\langle f, c', \text{ANY} \rangle\}$. This means that the resulting value from the variable read is the value $[\text{ANY} \mapsto \text{"bar"}]$, so we obtain the same precision as with heap specialization.

This first example shows how the free variable partitioning mechanism works and how it relates to heap specialization, but it does not demonstrate any precision improvements compared to the existing TAJJS analyzer, which does apply heap specialization. The second example, Figure 14, illustrates a simplified version of how free variable partitioning was used in the motivating example in combination with property name partitioning, which leads to a precision improvement of TAJJS. Line 32 defines the object `o1` with two properties, and line 33 defines `o2` as an empty object. Lines 34–41 iterate over the properties of `o1`. For each property, it writes a function returning the value of `o1[p]` to the `p` property of `o2`. To prove that the assertion at line 42 always holds, it is critical that the values of v are not mixed together in the iterations.

Using property name partitioning at line 36, the value of v becomes $[\text{VAL}\langle n, r, \text{"x"} \rangle \mapsto 1, \text{VAL}\langle n, r, \text{"y"} \rangle \mapsto 2]$ and the value of \mathbf{p} becomes $[\text{VAL}\langle n, r, \text{"x"} \rangle \mapsto \text{"x"}, \text{VAL}\langle n, r, \text{"y"} \rangle \mapsto \text{"y"}]$, where n is the read property node and r is the register storing the property name. (For clarity we ignore the OTHER partition in this example.) When analyzing the closure creation at line 37, we use the top rules in Figure 11 with $LC = \{\text{VAL}\langle n, r, \text{"x"} \rangle, \text{VAL}\langle n, r, \text{"y"} \rangle\}$. This means that v is augmented with the additional partitions $[\text{FUN}\langle \mathbf{h}, c, \text{VAL}\langle n, r, \text{"x"} \rangle \rangle \mapsto 1, \text{FUN}\langle \mathbf{h}, c, \text{VAL}\langle n, r, \text{"y"} \rangle \rangle \mapsto 2]$, and the value being written to `o2[p]` is $[\text{VAL}\langle n, r, \text{"x"} \rangle \mapsto (\{\text{FUN}\langle \mathbf{h}, c, \text{VAL}\langle n, r, \text{"x"} \rangle \rangle\}, \ell_j), \text{VAL}\langle n, r, \text{"y"} \rangle \mapsto (\{\text{FUN}\langle \mathbf{h}, c, \text{VAL}\langle n, r, \text{"y"} \rangle \rangle\}, \ell_j)]$. Here, ℓ_j denotes the location of the closure created in line 37. At the dynamic property write, the property name and value to be written share the covering $\{\text{VAL}\langle n, r, \text{"x"} \rangle, \text{VAL}\langle n, r, \text{"y"} \rangle\}$, meaning that the write happens as described in Figure 8, so that `o2.x` becomes $(\{\text{FUN}\langle \mathbf{h}, c, \text{VAL}\langle n, r, \text{"x"} \rangle \rangle\}, \ell_j)$ and `o2.y` becomes $(\{\text{FUN}\langle \mathbf{h}, c, \text{VAL}\langle n, r, \text{"y"} \rangle \rangle\}, \ell_j)$. Now when `o2.y` is called in line 42, the call to \mathbf{j} is augmented with the the set of function partitions $\{\text{FUN}\langle \mathbf{h}, c, \text{VAL}\langle n, r, \text{"y"} \rangle \rangle\}$. Therefore when reading the value v in line 38, according to Figure 12 we only read the $\text{FUN}\langle \mathbf{h}, c, \text{VAL}\langle n, r, \text{"y"} \rangle \rangle$ partition. The result of reading v is then $[\text{ANY} \mapsto 2]$, so the analysis is precise enough to prove that the assertion at line 42 holds.

5.3 Type Partitioning

Value partitioning can also be useful for partitioning values based on their types. Since JavaScript does not have function overloading, it is common to reflectively find the type of an argument, and based on the type run different pieces of code (as in line 3 in Figure 1).

16:18 Value Partitioning

$$\sigma(r_3) := \begin{cases} \sigma(r_3) \uplus [\text{TYPE}\langle n, r_3, ty \rangle \mapsto \text{FILTER}(\sigma(r_3), ty) \mid ty \in \text{TYPES}(\sigma(r_3))] & \text{if } |\text{TYPES}(\sigma(r_3))| > 1 \\ \sigma(r_3) & \text{otherwise} \end{cases}$$

■ **Figure 15** Addition to the transfer function for a call node with one argument, $r_1 \leftarrow r_2(r_3)$. `FILTER` restricts a partitioned value to represent only values that match the given type, and `TYPES` returns the possible types of the given partitioned value.

This is often done through the use of predicate functions, which are one-parameter functions that return a boolean value. By partitioning the arguments at calls to predicate functions, the analysis becomes able to track the relations between the arguments and the return values, and thereby boost the control sensitivity mechanism (see Section 3) at branches that involve such calls. Since the analysis does not know in advance whether a function returns boolean values, we simply perform this partitioning at all function calls with one argument, without considering what values the function may return.

Partition tokens for type partitioning

We introduce type partitioning tokens of the form `TYPE` $\langle n, r, ty \rangle$, where $n \in N$ is a call node $r_1 \leftarrow r_2(r_3)$, $r \in R$ is the argument register in n (in this case r_3), and $ty \in Types$ using the set of types shown in Figure 6. Such a token represents the set of traces where the type of r is ty at the last occurrence of n . For example, the traces that reach line 7 in Figure 1 are represented by the token `TYPE` $\langle n, r, \text{function} \rangle$ where n is the call to `isFunction` in line 6 and r is the argument register of that call node.

Function calls

Figure 15 shows an addition to the transfer function for call nodes, $r_1 \leftarrow r_2(r_3)$, to partition the argument value before the call takes place. The first case applies if the argument $\sigma(r_3)$ abstractly represents values of multiple types (i.e., $|\text{TYPES}(\sigma(r_3))| > 1$, where `TYPES` returns the set of all the types the given abstract value may have). In this case we introduce a partition `TYPE` $\langle n, r_3, ty \rangle$ for each $ty \in \text{TYPES}(\sigma(r_3))$, such that the value in that partition is `FILTER` $(\sigma(r_3), ty)$, where `FILTER` restricts $\sigma(r_3)$ to only represent values of type ty . Since all the possible types are represented, the new partitions together form a covering.

Recall that a `TYPE` $\langle n, r, ty \rangle$ token only represents information about the last occurrence of n in a given trace. To ensure this property we always remove all existing `TYPE` $\langle n, _, _ \rangle$ tokens from the abstract state immediately before applying the modified transfer function for call node n .

Example

As an example consider the code in Figure 16, and assume \mathbf{x} has the abstract value `fun1|obj2` (representing either the function `fun1` or the object `obj2`). Without type partitioning, the result of analyzing the `isObj(x)` call is the abstract value `AnyBool` (representing true or false), so both branches are analyzed with \mathbf{x} being `fun1|obj2`; however, in a concrete execution, `fun1` will never flow to the “true” branch, and `obj2` will never flow to the “false” branch.

```

43 function isObj(arg) {
44   return typeof arg == 'object';
45 }
46 if (isObj(x)) { ... } else { ... }

```

■ **Figure 16** Type partitioning example.

```

47 function isObj(arg) {
48   if (typeof arg == 'object')
49     return true;
50   else
51     return false;
52 }
53 if (isObj(x)) { ... } else { ... }

```

■ **Figure 17** Type partitioning example with control dependent relations.

By using type partitioning, we partition \mathbf{x} before calling the predicate function. In this example let n be the call node and let r be its argument register. Then \mathbf{x} becomes $[\text{TYPE}\langle n, r, \text{function} \rangle \mapsto \text{fun1}, \text{TYPE}\langle n, r, \text{object} \rangle \mapsto \text{obj2}]$. Now when analyzing the body of `isObj`, the expression `typeof arg == 'object'` evaluates to the partitioned value $[\text{TYPE}\langle n, r, \text{function} \rangle \mapsto \text{false}, \text{TYPE}\langle n, r, \text{object} \rangle \mapsto \text{true}]$. When reaching the `if` branch, control sensitivity ensures that only the `object` partition flows to the “true” branch (i.e., \mathbf{x} 's value becomes $[\text{TYPE}\langle n, r, \text{function} \rangle \mapsto \perp, \text{TYPE}\langle n, r, \text{object} \rangle \mapsto \text{obj2}]$ in that branch), and only the `function` partition flows to the “false” branch (i.e., \mathbf{x} 's value becomes $[\text{TYPE}\langle n, r, \text{function} \rangle \mapsto \text{fun1}, \text{TYPE}\langle n, r, \text{object} \rangle \mapsto \perp]$ in that branch).

Control dependent relations

Predicate functions are sometimes implemented with control dependent relations between the argument and the result, as in the example in Figure 17. The example is contrived but it is not uncommon in predicate functions that the result values appear as the literals `true` or `false` in branches. With the type partitioning mechanism described above, the returned values will not be partitioned in this situation, since the partitions in `arg` do not propagate to the values `true` and `false`.

To mitigate this issue, we augment the abstract states as shown in Figure 18 to keep track of partitions that must be dead or may be live (represented by the two $\mathcal{P}(T)$ components, respectively). A partition is *dead* if the set of traces it represents is empty, and it is live otherwise. (We only keep track of the live partitions in coverings where there are any dead partitions.) Since the branch condition `typeof arg == 'object'` is analyzed with a partitioned value for `arg`, by control sensitivity we know that the only traces that can reach the “true” branch are those represented by the `object` partition, so we record that $\text{TYPE}\langle n, r, \text{object} \rangle$ is live and $\text{TYPE}\langle n, r, \text{function} \rangle$ is dead in that branch, and conversely in the other branch. To exploit this information, we also update the transfer function for constants, $r_1 \leftarrow c$, as shown in Figure 19. Basically, it assigns \perp to all dead partitions and the constant c to all live partitions. If there are no dead partitions, it behaves as usual, where the constant is written to the ANY partition. When the analysis reaches `true` (line 49), we obtain the partitioned value $[\text{TYPE}\langle n, r, \text{function} \rangle \mapsto \perp, \text{TYPE}\langle n, r, \text{object} \rangle \mapsto \text{true}]$, and similarly when analyzing `false` (line 51) we get $[\text{TYPE}\langle n, r, \text{function} \rangle \mapsto \text{false}, \text{TYPE}\langle n, r, \text{object} \rangle \mapsto \perp]$. The join of these two values is $[\text{TYPE}\langle n, r, \text{function} \rangle \mapsto \text{false}, \text{TYPE}\langle n, r, \text{object} \rangle \mapsto \text{true}]$, which becomes the result of `isObj(x)`. Due to the control sensitivity mechanism, only `obj2` then flows to the “true” branch, and only `fun1` flows to the “false” branch in line 53.

$$State' = State \times \mathcal{P}(T) \times \mathcal{P}(T)$$

■ **Figure 18** Abstract states updated to keep track of dead and live partitions.

$$\sigma(r_1) := \begin{cases} [t \mapsto c \mid t \in \text{LIVEPARTITIONS}(\sigma)] \uplus [t \mapsto \perp \mid t \in \text{DEADPARTITIONS}(\sigma)] & \text{if } \text{DEADPARTITIONS}(\sigma) \neq \emptyset \\ [ANY \mapsto c] & \text{otherwise} \end{cases}$$

■ **Figure 19** Updated transfer function for constant nodes, $r_1 \leftarrow c$, for improved type partitioning.

6 Evaluation

We have implemented the value partitioning framework (Section 4) and the three instantiations (Section 5) on top of TAJJS v0.24. Implementing the general framework in TAJJS required 900 lines of code, however most of this is boilerplate code for lifting operations on ordinary abstract values to also work on partitioned values. With the general framework in place, instantiations are easy to implement: property name partitioning (Section 5.1), free variable partitioning (Section 5.2), and type partitioning (Section 5.3) required only around 230, 250, and 60 lines of code, respectively. We disable TAJJS’s **for-in** specialization technique, since it is subsumed by property name partitioning.¹⁷ We refer to our new analysis tool as TAJJS_{VALPAR}.¹⁸ Using this tool we evaluate our techniques by answering the following research questions:

RQ1 How does TAJJS_{VALPAR} compare to existing state-of-the-art analyses for JavaScript?

RQ2 What are the effects of the three different instantiations of value partitioning?

All our experiments are conducted on an Ubuntu machine with a 2.6 GHz Intel Xeon E5-2697A CPU running a JVM with 10 GB RAM.

6.1 RQ1: Comparison with State-Of-The-Art Analyses

We start by comparing TAJJS_{VALPAR} against the current state-of-the-art analyses for JavaScript: the baseline TAJJS analyzer with static determinacy [2], TAJJS_{VR} [26] with demand-driven value refinement, and the CompAbs analyzer [16] based on the SAFE analyzer [17]. We use the same benchmarks as those used in the evaluation of TAJJS_{VR}, which is the most recent related work.

Micro benchmarks

We first evaluate TAJJS_{VALPAR} against a small collection of micro benchmarks that capture some of the main challenges that appear in analysis of modern JavaScript libraries and are used in previous work [16, 26]. The benchmarks all contain dynamic read/write pairs that

¹⁷The motivation for introducing **for-in** specialization in [2] was to reason about correlated read/write pairs inside **for-in** loops. This relational information is now provided by property name partitioning.

¹⁸TAJJS_{VALPAR}: TAJJS with Value Partitioning

■ **Table 1** Micro-benchmarks that check how state-of-the-art analyses handle various dynamic read/write pairs that represent typical challenges in JavaScript library code. A ✗ indicates that the analysis mixes together the properties of the object being manipulated, while a ✓ indicates that it is sufficiently precise to keep them distinct. The CF, CG, AF, and AG benchmarks are drawn directly from [16], while M1, M2, and M3 are drawn directly from [26].

Benchmark	TAJS	CompAbs	TAJS _{VR}	TAJS _{VALPAR}
CF	✓	✓	✓	✓
CG	✓	✓	✓	✓
AF	✗	✓	✓	✓
AG	✗	✓	✓	✓
M1	✗	✗	✓	✓
M2	✗	✗	✓	✓
M3	✗	✗	✓	✓

are variations of the pattern shown in the introduction and the motivating example. The results of the comparison are shown in Table 1. For these benchmarks, a test *succeeds* if it avoids mixing together properties in the dynamic read/write pairs.

The first two examples, CF and CG, are loops where the static analyses have enough information to be able to unroll all the iterations and thereby analyze the read/write patterns with precise property names. For CF, property name partitioning in TAJS_{VALPAR} gives the same degree of precision without loop unrolling.

AF and AG are loops where the static analyses are incapable of obtaining a precise value for the property name used in the dynamic read/write pairs. TAJS fails to analyze these, but CompAbs detects the pattern syntactically and therefore applies trace partitioning to analyze the code precisely. TAJS_{VR} also succeeds on these tests, because its backwards abstract interpreter is capable of providing the necessary relational information. In comparison, TAJS_{VALPAR} can reason about the relational information on its own.

Both TAJS and CompAbs fail on the last three tests (M1, M2, and M3). CompAbs fails on M1 and M3 because it does not apply partitioning due to the fragility of syntactic patterns, and it fails on M2 because the partitioning does not provide the necessary precision about free variables. Again, TAJS_{VR} can analyze them all, since the backwards abstract interpreter is powerful enough to reason about all the cases, whereas TAJS_{VALPAR} successfully preserves the relational properties by the use of value partitioning.

These results demonstrate that for these benchmarks, TAJS_{VALPAR} is capable of providing comparable precision to the demand-driven value refinement technique without the need for a complicated backwards analysis, and provides better precision than the other analyses.

Library benchmarks

The next set of benchmarks is taken from the evaluation of TAJS_{VR} and consists of small test cases for popular real-world libraries. The libraries include the widely used functional utility library Underscore (which has more than 20 000 dependents in npm) v1.8.3 with 1 548 LoC and the most depended-upon package Lodash (more than 115 000 dependents). We analyze both Lodash3 (v3.0.0, 10 785 LoC) and Lodash4 (v4.17.10, 17 105 LoC), since their code bases are substantially different and therefore pose distinct challenges for static analysis.

■ **Table 2** Analysis results for real-world benchmarks (from [26]). For each group of benchmarks and for each of the four analyzers, we show the number of tests that are analyzed successfully and (in parentheses) the average analysis time per successful test.

Benchmark group	TAJS	CompAbs	TAJS _{VR}	TAJS _{ValPar}
Underscore (182 tests)	0 (-)	0 (-)	173 (2.9s)	173 (2.7s)
Lodash3 (176 tests)	7 (2.4s)	0 (-)	172 (5.5s)	173 (5.3s)
Lodash4 (306 tests)	0 (-)	0 (-)	266 (24.7s)	289 (26.3s)
Prototype (6 tests)	0 (-)	2 (23.1s)	5 (97.7s)	5 (34.1s)
Scriptaculous (1 tests)	0 (-)	1 (62.0s)	1 (236.9s)	1 (55.2s)
jQuery (71 tests)	3 (16.0s)	0 (-)	3 (13.5s)	3 (20.4s)

The other libraries, Prototype v1.7.2, Scriptaculous v1.9.0, and jQuery v1.10,¹⁹ are popular libraries for client-side web programming.

The analysis results are shown in Table 2. We classify an analysis of a benchmark as successful if it terminates within 5 minutes and the analysis result to our knowledge is sound. In particular, an analysis run is considered a failure if the analysis result does not have dataflow to the ordinary exit of the program. (All the tests pass in normal execution, so an analysis result is obviously unsound if there is no dataflow to the ordinary exit.) To increase confidence in the soundness of the analysis results for TAJS_{VALPAR}, we apply thorough soundness testing as described at the end of this section. Increasing the time budget does not help for these benchmarks: as reported previously for JavaScript analysis tools, critical precision losses tend to cause a proliferation of spurious dataflow that drastically increases analysis time and renders the analysis results useless [26, 11, 22, 16].

The results for TAJS_{VALPAR} are comparable to those of TAJS_{VR}, which outperforms the other analyzers. TAJS_{VALPAR} succeeds in analyzing all the benchmarks that TAJS_{VR} can handle, plus 24 more (one Lodash3 test and 23 Lodash4 tests). Note the substantial improvement for the Lodash4 tests: the number of Lodash4 tests that are not analyzed successfully is reduced from 40 to 17. None of the analyzers do well on the jQuery benchmarks; a preliminary manual study shows that the reasons are unrelated to relational analysis. The results are as expected, since property name partitioning and free variable partitioning are alternative techniques to provide the relational information that TAJS_{VR} obtains from its demand-driven value refinement. Furthermore, value partitioning is triggered more often during the analysis, which means that the precision improvements are not limited to the few critical cases where value refinement is triggered. On top of this, type partitioning provides some additional precision beyond the capabilities of TAJS_{VR}.

Comparing the performance between TAJS_{VALPAR} and TAJS_{VR}, the most significant differences are for the Prototype and Scriptaculous benchmarks. TAJS_{VALPAR} is around 3–4 times faster than TAJS_{VR}, which is mainly because property name partitioning makes the **for-in** specialization technique in TAJS obsolete. For Underscore and Lodash3, TAJS_{VALPAR} is slightly faster than TAJS_{VR}. This is encouraging, because analyzing dynamic property writes as the one in line 7 in Figure 1 is more expensive in TAJS_{VALPAR} than in TAJS_{VR}. In TAJS_{VR} such an operation is handled as a single imprecise write (since the precision is recovered on demand), whereas TAJS_{VALPAR} performs the write for each property that is copied. To soundly handle setters, all the writes happen in different states that are

¹⁹This is the version of jQuery used in [2]. Note that [16] used the older v1.4.4.

subsequently joined together, which causes $\text{Tajs}_{\text{VALPAR}}$ to spend some extra time at such writes. Since the analysis time is nevertheless similar, we can conclude that value partitioning is cheaper for analyzing other parts of the libraries. For `Lodash4` and `jQuery`, Tajs_{VR} is slightly faster than $\text{Tajs}_{\text{VALPAR}}$. For `Lodash4`, the main reason is the handling of dynamic property writes, and for the `jQuery` benchmarks, type partitioning adds little performance overhead as seen in Table 3.

Precision

Previous work [2, 22, 26] established that type analysis and call-graph construction are useful metrics for measuring the precision of an analysis for JavaScript, and therefore we use these metrics to evaluate the analysis precision of $\text{Tajs}_{\text{VALPAR}}$. All locations are treated context-sensitively in these measurements, meaning that we count the same location once for each reachable context. We count the number of possible types for the resulting value in each variable or property read and find that in 99.19% of the reads, a single unique type is read, with the average number of types being 1.02. For measuring precision of the call-graph construction, we measure the number of call-sites with unique callees, and find this number to be 99.95% of all call-sites. These numbers show that when the analysis succeeds, it does so with very high precision.

Soundness

Formally proving soundness of the three variants of value partitioning is out of scope of this paper, however, we will informally justify that the general approach is sound. Since general trace partitioning is known to be sound, it suffices to argue that the precision gained by value partitioning is equivalent to that obtained through trace partitioning. The key reason why this holds for property name partitioning and type partitioning is that the partition tokens represent the last occurrence of some node, meaning that if two values share partitions, they represent information about the same execution traces. This means that we could (if ignoring performance) instead have applied traditional trace partitioning, with exactly the same partition tokens and at the same nodes, resulting in the same precision. (For further discussion about the connection between value partitioning and trace partitioning, see Section 7.) Similarly for free variable partitioning, since the partitions are only allowed on activation objects, the precision is never higher than what would be obtained using heap specialization (where each partition would be represented by a distinct abstract activation object), and therefore soundness follows from soundness of heap specialization.

Furthermore, to increase confidence in the soundness of our implementation, all the $\text{Tajs}_{\text{VALPAR}}$ results have been thoroughly soundness tested [3]. This means that the analysis results overapproximate all the dataflow facts that have been observed during concrete executions of the analyzed benchmarks. For every variable and property read observed concretely, we have checked that the concrete value is in the concretization of the corresponding abstract value in the analysis results, and similarly for property writes and function calls. All our benchmarks except one pass in total more than 7.6 million soundness tests. The one benchmark that fails is a `Lodash4` test, which uses ES6 iterators in combination with `Arrays.from`, which is not fully supported in the latest version of `Tajs` and is unrelated to the use of value partitioning.

■ **Table 3** Analysis results for real-world benchmarks (from [26]) using different instantiations of value partitioning. “None” is without value partitioning, “P” is with property name partitioning, “P + FV” is with property name and free variable partitioning, and “F + PV + T” is with property name, free variable, and type partitioning.

Benchmark group	None	P	P + FV	P + FV + T
Underscore (182 tests)	0 (-)	149 (2.0s)	173 (2.5s)	173 (2.7s)
Lodash3 (176 tests)	7 (2.4s)	167 (4.7s)	173 (5.1s)	173 (5.3s)
Lodash4 (306 tests)	0 (-)	268 (16.8s)	274 (27.7s)	289 (26.3s)
Prototype (6 tests)	0 (-)	0 (-)	5 (32.7s)	5 (34.1s)
Scriptaculous (1 tests)	0 (-)	0 (-)	1 (53.1s)	1 (55.2s)
jQuery (71 tests)	3 (16.0s)	3 (15.2s)	3 (16.5s)	3 (20.4s)

6.2 RQ2: Effects of the Three Instantiations

We now investigate how much each of the three uses of value partitioning contributes to the results reported in the previous section. The results from running our analysis with only some instantiations enabled can be seen in Table 3. The column “P” is with only property name partitioning enabled; we see that it is sufficient for analyzing many of the Underscore and Lodash test cases, but not for any of the Prototype or Scriptaculous test cases. (Without property name partitioning but with the other two instantiations enabled, the analysis is not able to analyze more benchmarks than TAJ.S.) The column “P + FV” uses both property name partitioning and free variable partitioning. Also enabling free variable partitioning makes the analysis capable of analyzing many additional benchmarks: more Underscore and Lodash test cases, as well as some Prototype and Scriptaculous test cases. Compared to only property name partitioning, the analysis times are higher (for the reason discussed above regarding additional state joins). The last column “P + FV + T” is with all instantiations enabled and therefore contains the same numbers as shown in Table 2. We see that type partitioning enables the analysis of 15 additional Lodash4 tests, without significantly increasing the analysis time.

We conclude that all three instantiations contribute to the results, where property name partitioning is the most important one, followed by free variable partitioning and then type partitioning. (TAJS already performs filtering at branches, as mentioned in Section 3; without that feature the effect of type partitioning would likely be larger.)

7 Related Work

Trace partitioning

Value partitioning can be viewed as a variant of trace partitioning [23] as explained in Sections 1, 2 and 4, but there are some important differences. Changing the original abstract domain in Section 4 to support traditional trace partitioning can be done by replacing $L \rightarrow State$ by $L \rightarrow T \rightarrow State$, so that an abstract state is maintained for each partition, at every location. Thus, different locations can partition the abstract states differently. Value partitioning instead has only one abstract state per location but partitions the individual abstract values, which adds an additional degree of flexibility: different parts of each abstract state can be partitioned differently. In particular, for the large parts of the states where we

are not interested in relational information, we can use the $\{\text{ANY}\}$ partitioning,²⁰ while for the important registers and object properties, we can have nontrivial partitions. With traditional trace partitioning, the normal transfer functions are applied for each partition, which causes redundant computations because of the similarities between the different partitions²¹; with value partitioning, we only pay a price for partitioning at operations that involve abstract values with nontrivial partitions. This is the main reason for the low overhead of the technique.

Another difference is that the partition tokens in traditional trace partitioning are actually lists of “directives” (the language of directives used by Rival and Mauborgne [23] is similar to our language of tokens in Figure 6), which can lead to a combinatorial explosion. By partitioning at the level of values and allowing multiple coverings in each partitioned value, we avoid the need to maintain such combinations.

Relational analysis

Traditional techniques for achieving relational analysis, as exemplified by the octagon abstract domain [19], focus on numeric relations, such as, linear inequalities. To reduce the cost of this approach, a syntactic pre-analysis called variable packing is typically used for partitioning the set of program variables, and one octagon is then used for each pack instead of tracking all possible combinations of inequalities. This kind of partitioning is reminiscent of value partitioning, but with the important difference that variable packing and octagons operate on sets of program variables whereas value partitioning works on individual abstract values. In our work with analysis of JavaScript libraries, we have not encountered a critical need for tracking numeric relations.

The well-known analyzer Astrée [4] applies not only trace partitioning and octagons, but also a decision tree abstract domain that is used for tracking relations between booleans and numerical variables that affect control flow. That technique has some similarities with our type partitioning mechanism but relies on variable packing to avoid combinatorial explosions, whereas type partitioning uses the more lightweight value partitioning technique in combination with the existing control sensitivity mechanism of TAJs.

The main purpose of value partitioning is to be able to reason about relations between different parts of the abstract state (i.e., program variables and registers) at the various program points. Some literature uses the term relational analysis with a slightly different meaning: to relate information across program points, typically relations between the entry and exits of functions [8, 5].

Static analysis for JavaScript

Through the last decade, several static analyzers for JavaScript have been developed, including WALA [25, 24, 28], SAFE [17, 22], JSAI [13], and TAJs [11, 2, 26]. Although we focus on TAJs, the designs of SAFE and JSAI are reasonably similar, so we believe value partitioning could also be incorporated into those tools with little effort.

As discussed in the introduction, much work has been put into improving precision of the analyses through different kinds of context sensitivity and elaborate abstract domains. The techniques include parameter sensitivity and heap context sensitivity [2], loop unrolling [22],

²⁰ In our experiments, 99.4% of all abstract values have the trivial $\{\text{ANY}\}$ partitioning.

²¹ This was shown experimentally in the work on TAJ_{SVR} [26, Section 7.1].

and syntactic patterns for detecting correlated read/write pairs and guiding context sensitivity [25]. Other works have explored more expressive string abstractions to reason more precisely about property names in dynamic property accesses [18, 1, 21]. Our abstract domain extension for value partitioning has few assumptions about the underlying abstract domain, so most of these techniques can be combined with value partitioning.

Despite such precision improvement techniques, imprecision is inevitable, and only a few techniques have been designed to handle dynamic property accesses with imprecise property names, most importantly, CompAbs-style trace partitioning [16] and demand-driven value refinement [26]. Previous work has shown that demand-driven value refinement enables analysis of many more challenging benchmarks than CompAbs-style trace partitioning (as also discussed in Section 6), and that the trace partitioning approach causes a large amount of redundant computation [26, Section 7.1]. The fundamental drawback of demand-driven value refinement is that it requires a separate backwards abstract interpreter for not only the entire JavaScript language but also the standard library. The backwards abstract interpreter of TAJSV_R is not simply the dual of TAJ_S but works goal-directed and with its own abstract domain based on intuitionistic separation logic. In contrast, value partitioning directly leverages the existing forward analyzer and thereby supports both the JavaScript language and the standard library essentially for free, which makes this approach substantially easier to develop and maintain. Furthermore, value partitioning is more general (for example, it enables type partitioning), and the three instantiations we have presented lead to better precision (for the Lodash4 tests).

The HOO (heap with open objects) abstract domain [9] is a relational abstraction that is designed to reason more precisely about abstract objects whose properties cannot be known statically. That approach is highly expressive but not scalable to real-world JavaScript libraries as those considered in Section 6.

8 Conclusion

We have presented value partitioning, a static analysis technique for reasoning about relational properties. It is a lightweight alternative to traditional trace partitioning techniques that allows relational information to be incorporated into the abstract values instead of requiring separate abstract states for the partitions. We have proposed three instantiations of value partitioning in JavaScript analysis: property name partitioning, free variable partitioning, and type partitioning, which enable precise reasoning for dynamic read/write pairs, free variables, and predicate functions, respectively.

The experimental results show that extending the TAJ_S analyzer with the three variants of value partitioning enables precise and efficient analysis of complex JavaScript libraries including Lodash and Underscore, thereby outperforming a state-of-the-art technique that relies on trace partitioning and without requiring a complicated backwards analysis. For the libraries considered in this study, property name partitioning has the largest effect among the proposed variants.

An interesting direction for future research is to investigate whether some of the traditional context sensitivity strategies used in TAJ_S and other JavaScript analyzers can be reformulated as new value partitioning instantiations, to make analysis faster while retaining precision.

References


- 1 Roberto Amadini, Alexander Jordan, Graeme Gange, François Gauthier, Peter Schachte, Harald Søndergaard, Peter J. Stuckey, and Chenyi Zhang. Combining string abstract domains for JavaScript analysis: An evaluation. In *Tools and Algorithms for the Construction and Analysis of Systems - 23rd International Conference, TACAS 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings, Part I*, volume 10205 of *Lecture Notes in Computer Science*, pages 41–57, 2017.
- 2 Esben Andreasen and Anders Møller. Determinacy in static analysis for jQuery. In *Proc. ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 17–31, 2014.
- 3 Esben Sparre Andreasen, Anders Møller, and Benjamin Barslev Nielsen. Systematic approaches for increasing soundness and precision of static analyzers. In *Proceedings of the 6th ACM SIGPLAN International Workshop on State Of the Art in Program Analysis, SOAP@PLDI 2017, Barcelona, Spain, June 18, 2017*, pages 31–36, 2017.
- 4 Bruno Blanchet, Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. A static analyzer for large safety-critical software. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation 2003, San Diego, California, USA, June 9-11, 2003*, pages 196–207. ACM, 2003.
- 5 Cristiano Calcagno, Dino Distefano, Peter W. O’Hearn, and Hongseok Yang. Compositional shape analysis by means of bi-abduction. *J. ACM*, 58(6):26:1–26:66, 2011.
- 6 David R. Chase, Mark N. Wegman, and F. Kenneth Zadeck. Analysis of pointers and structures. In *Proceedings of the ACM SIGPLAN’90 Conference on Programming Language Design and Implementation (PLDI), White Plains, New York, USA, June 20-22, 1990*, pages 296–310. ACM, 1990.
- 7 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, POPL ’77*, pages 238–252, New York, NY, USA, 1977. ACM.
- 8 Patrick Cousot and Radhia Cousot. Modular static program analysis. In *Compiler Construction, 11th International Conference, CC 2002, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2002, Grenoble, France, April 8-12, 2002, Proceedings*, volume 2304 of *Lecture Notes in Computer Science*, pages 159–178. Springer, 2002.
- 9 Arlen Cox, Bor-Yuh Evan Chang, and Xavier Rival. Automatic analysis of open objects in dynamic language programs. In *Static Analysis - 21st International Symposium, SAS 2014, Munich, Germany, September 11-13, 2014. Proceedings*, pages 134–150, 2014.
- 10 Arjun Guha, Claudiu Saftoiu, and Shriram Krishnamurthi. Typing local control and state using flow analysis. In *Programming Languages and Systems - 20th European Symposium on Programming, ESOP 2011, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2011, Saarbrücken, Germany, March 26-April 3, 2011. Proceedings*, volume 6602 of *Lecture Notes in Computer Science*, pages 256–275. Springer, 2011.
- 11 Simon Holm Jensen, Anders Møller, and Peter Thiemann. Type analysis for JavaScript. In *Proc. 16th International Static Analysis Symposium*, pages 238–255, 2009.
- 12 John B. Kam and Jeffrey D. Ullman. Monotone data flow analysis frameworks. *Acta Inf.*, 7(3):305–317, September 1977.
- 13 Vineeth Kashyap, Kyle Dewey, Ethan A. Kuefner, John Wagner, Kevin Gibbons, John Sarracino, Ben Wiedermann, and Ben Hardekopf. JSAI: a static analysis platform for JavaScript. In *Proc. 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 121–132, 2014.

- 14 Vineeth Kashyap, John Sarracino, John Wagner, Ben Wiedermann, and Ben Hardekopf. Type refinement for static analysis of JavaScript. In *DLS'13, Proceedings of the 9th Symposium on Dynamic Languages, part of SPLASH 2013, Indianapolis, IN, USA, October 26-31, 2013*, pages 17–26. ACM, 2013.
- 15 Gary A. Kildall. A unified approach to global program optimization. In *Conference Record of the ACM Symposium on Principles of Programming Languages, Boston, Massachusetts, USA, October 1973*, pages 194–206. ACM Press, 1973.
- 16 Yoonseok Ko, Xavier Rival, and Sukyoung Ryu. Weakly sensitive analysis for JavaScript object-manipulating programs. *Softw., Pract. Exper.*, 49(5):840–884, 2019.
- 17 Hongki Lee, Sooncheol Won, Joonho Jin, Junhee Cho, and Sukyoung Ryu. SAFE: formal specification and implementation of a scalable analysis framework for ECMAScript. In *Proc. International Workshop on Foundations of Object Oriented Languages*, 2012.
- 18 Magnus Madsen and Esben Andreasen. String analysis for dynamic field access. In *Proc. 23rd International Conference on Compiler Construction*, volume 8409 of *Lecture Notes in Computer Science*. Springer, 2014.
- 19 Antoine Miné. The octagon abstract domain. *Higher-Order and Symbolic Computation*, 19(1):31–100, 2006.
- 20 Erik M. Nystrom, Hong-Seok Kim, and Wen-mei W. Hwu. Importance of heap specialization in pointer analysis. In *Proceedings of the 2004 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis For Software Tools and Engineering, PASTE'04, Washington, DC, USA, June 7-8, 2004*, pages 43–48. ACM, 2004.
- 21 Changhee Park, Hyeonseung Im, and Sukyoung Ryu. Precise and scalable static analysis of jquery using a regular expression domain. In *Proceedings of the 12th Symposium on Dynamic Languages, DLS 2016*, pages 25–36, New York, NY, USA, 2016. ACM.
- 22 Changhee Park and Sukyoung Ryu. Scalable and precise static analysis of JavaScript applications via loop-sensitivity. In *Proc. 29th European Conference on Object-Oriented Programming*, pages 735–756, 2015.
- 23 Xavier Rival and Laurent Mauborgne. The trace partitioning abstract domain. *ACM Trans. Program. Lang. Syst.*, 29(5), August 2007.
- 24 Max Schäfer, Manu Sridharan, Julian Dolby, and Frank Tip. Dynamic determinacy analysis. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13, Seattle, WA, USA, June 16-19, 2013*, pages 165–174. ACM, 2013.
- 25 Manu Sridharan, Julian Dolby, Satish Chandra, Max Schäfer, and Frank Tip. Correlation tracking for points-to analysis of JavaScript. In *Proc. 26th European Conference on Object-Oriented Programming*, 2012.
- 26 Benno Stein, Benjamin Barslev Nielsen, Bor-Yuh Evan Chang, and Anders Møller. Static analysis with demand-driven value refinement. *Proceedings of the ACM on Programming Languages (PACMPL)*, 3:140:1–140:29, 2019.
- 27 Mark N. Wegman and F. Kenneth Zadeck. Constant propagation with conditional branches. *ACM Trans. Program. Lang. Syst.*, 13(2):181–210, 1991.
- 28 Shiyi Wei, Omer Tripp, Barbara G. Ryder, and Julian Dolby. Revamping JavaScript static analysis via localization and remediation of root causes of imprecision. In *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2016, Seattle, WA, USA, November 13-18, 2016*, pages 487–498. ACM, 2016.


Static Type Analysis by Abstract Interpretation of Python Programs

Raphaël Monat 

Sorbonne Université, CNRS, LIP6, Paris, France
raphael.monat@lip6.fr

Abdelraouf Ouadjaout 

Sorbonne Université, CNRS, LIP6, Paris, France
abdelraouf.ouadjaout@lip6.fr

Antoine Miné 

Sorbonne Université, CNRS, LIP6, Paris, France
Institut Universitaire de France, Paris, France
antoine.mine@lip6.fr

Abstract

Python is an increasingly popular dynamic programming language, particularly used in the scientific community and well-known for its powerful and permissive high-level syntax. Our work aims at detecting statically and automatically type errors. As these type errors are exceptions that can be caught later on, we precisely track all exceptions (raised or caught). We designed a static analysis by abstract interpretation able to infer the possible types of variables, taking into account the full control-flow. It handles both typing paradigms used in Python, nominal and structural, supports Python's object model, introspection operators allowing dynamic type testing, dynamic attribute addition, as well as exception handling. We present a flow- and context-sensitive analysis with special domains to support containers (such as lists) and infer type equalities (allowing it to express parametric polymorphism). The analysis is soundly derived by abstract interpretation from a concrete semantics of Python developed by Fromherz et al. Our analysis is designed in a modular way as a set of domains abstracting a concrete collecting semantics. It has been implemented into the MOPSA analysis framework, and leverages external type annotations from the Typedsh project to support the vast standard library. We show that it scales to benchmarks a few thousand lines long, and preliminary results show it is able to analyze a small real-life command-line utility called PathPicker. Compared to previous work, it is sound, while it keeps similar efficiency and precision.

2012 ACM Subject Classification Theory of computation → Program analysis; Software and its engineering → Semantics

Keywords and phrases Formal Methods, Static Analysis, Abstract Interpretation, Type Analysis, Dynamic Programming Language, Python Semantics

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2020.17

Supplementary Material ECOOP 2020 Artifact Evaluation approved artifact available at <https://doi.org/10.4230/DARTS.6.2.11>.

Funding This work is partially supported by the European Research Council under Consolidator Grant Agreement 681393 – Mopsa.

Acknowledgements We thank the anonymous reviewers for their valuable comments and feedback.



© Raphaël Monat, Abdelraouf Ouadjaout, and Antoine Miné;
licensed under Creative Commons License CC-BY

34th European Conference on Object-Oriented Programming (ECOOP 2020).

Editors: Robert Hirschfeld and Tobias Pape; Article No. 17; pp. 17:1–17:29

Leibniz International Proceedings in Informatics



LIPICs Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany



1 Introduction

Sound static analyses for static languages, such as C and Java, are now widespread [5, 25, 11, 10, 33]. They have been particularly successful in the verification of critical embedded software, which use a subset of C and preclude complex control flow and dynamic memory allocation. The sound, efficient, and precise analysis of more dynamic languages and program traits remains a challenge.

Dynamic programming languages, such as JavaScript and Python, have become increasingly popular over the last years. Python currently ranks as the second most used programming language on Github.¹ It is appreciated for its powerful and permissive high-level syntax, e.g., it allows programmers to redefine all operators (addition, field access, etc.) in custom classes, and comes equipped with a vast standard library. Python is a highly dynamic language. It notably features:

1. *Dynamic typing*: variables in Python are neither statically declared nor typed. Any variable can hold a value of any type, and the type of the value held may change during program execution.
2. *Introspection*: programs can inspect the type of variables at run-time to alter their execution. Operators exist to support both nominal and structural types. Firstly, `isinstance(o, cls)` checks whether `o` has been instantiated from class `cls`, or a class inheriting from `cls`. For example, `isinstance("a", str)` returns `True`, while `isinstance(object(), str)` returns `False`. Secondly, `hasattr(o, attr)` checks whether `o` (or its class, or one of its parent class) has an attribute with name equal to the `attr` string. For instance, `hasattr(42, "__add__")` returns `True` because `int`, the class of `42`, has an addition method, named `"__add__"`. This kind of structural typing is so-called “duck typing”. Python classes are first-class objects and can be stored into variables.
3. *Self-modification*: the structure of Python objects can be altered at run-time. For instance, it is possible to add attributes to an object or class at run-time, or remove them. It is also possible to create new classes at run-time.
4. *Eval*: it is possible to evaluate arbitrary string expressions as Python code at run-time with `eval` statements.

Due to dynamic typing (point 1), type errors are detected at run-time and cause `TypeError` exceptions, whereas such errors would be caught at compile time in a statically typed language. In this article, we propose a static analysis to infer type information, and use this information to detect all exceptions that can be raised and not caught. Our type analysis is flow-sensitive, to take into account the fact that variable types evolve during program execution and, conversely, run-time type information is used to alter the control-flow of the program, either through introspection or method and operator overloading (points 2 and 3). Moreover, it is context-sensitive as, without any type information on method entry, it is not possible to infer its control flow at all. However, handling `eval` is left as future work (possibly leveraging ideas proposed by [17] on JavaScript).

Motivating Example. Consider the code from Fig. 1 (where `*` stands for a non-deterministic boolean). It defines a function `fspath`, taken from the standard library, with a parameter `p` as input. If `p` is an object instantiated from a class inheriting from `str` or `bytes`, it is returned. Otherwise, the function searches for an attribute called `fspath`, and calls it as a

¹ <https://octoverse.github.com/#top-languages>

```

1  def fspath(p):
2      sb = (str, bytes)
3      if isinstance(p, sb):
4          return p
5      if hasattr(p, "fspath"):
6          r = p.fspath()
7          if isinstance(r, sb):
8              return r
9          else: raise TypeError
10         else: raise TypeError
11
12     class Path:
13         def fspath(self):
14             return 42
15
16     if *: p = "/dev"
17     else: p = Path()
18     r = fspath(p)

```

■ **Figure 1** Motivating example.

method. If the return type is not an instance of `str` or `bytes`, an exception is raised. Thus, when `fspath` does not raise an error, it takes as input an instance of `str` or `bytes`, or an object having a method `fspath` returning either a string or a bytes-based string. In all cases, the return type of `fspath` is either `str` or `bytes`.

We model correct and erroneous calls to `fspath` in lines 12 to 18. In particular, we define a `Path` class, having a method `fspath` returning an integer, hence, a call to function `fspath` on an instance of `Path` would raise a `TypeError`. Our analysis is able to infer that, at the end of line 16, `p` is a string, and that at line 17, `p` is an instance of the `Path` class, which has a field `fspath` that can be called. It finds that, either `r` is a string, or a `TypeError` is raised.

As it is part of the standard library, the `fspath` function is particularly well-behaved: it does not make many implicit assumptions on the argument type (only that `p.fspath` is callable), but instead uses type information to handle different cases in different program branches. Nevertheless, the context is important to infer whether a specific call to `fspath` can raise a `TypeError` or not. A more typical Python programmer might replace lines 5 to 10 with a call to `return p.fspath()`, leaving implicit the fact that `p` should have a method `fspath` returning `str` or `bytes` chains. This is summarized in one of Python mottos: “easier to ask for forgiveness than permission”. Our analysis would correctly infer that invalid calls to that modified function would raise an `AttributeError` exception.

Static Analysis of Python. The sound analysis of Python programs is particularly challenging as it is a large and complex language. Python is not defined by a standard (formal or informal), but by its reference implementation, CPython. Fromherz et al. [13] introduced formally a concrete collecting semantics defined by structural induction on programs as a function manipulating reachable program states. We rely on a slight extension of this semantics. We then build a computable static analysis by abstract interpretation [9]. While [13] presented a value-analysis (employing in particular numeric domains), we introduce a more abstract analysis that only reasons on Python types. Similarly to [13], it tracks precisely the control-flow, including the flows of exceptions. We believe (and our preliminary experiments support) that flow-sensitive, context-sensitive type analysis for Python achieves a sweet-spot in term of efficiency versus usefulness. We believe that this level of precision is both necessary and, in practice, sufficient to infer the type-directed control flow (such as type testing or method dispatch) and infer exception propagation. Although our abstract domain tracks types precisely (including nominal types and attributes), it is nevertheless more scalable than a value analysis.

Relationship with Typing. It is worth comparing our approach with classic typing. Statically typed languages ensure the absence of type-related errors through static type checking, possibly augmented with automatic type inference. However, while static typing rejects untypable programs, our analysis gives a semantics to such programs by propagating type errors as exceptions. This is important in order to support programs that perform run-time type errors and catch them afterwards, which is common-place in Python. Indeed, our goal is not to enforce a stricter, easier to check, way to program in Python,² but rather to check as-is programs with no uncaught exceptions. Secondly, static type checking is generally flow-insensitive and context-insensitive, trying to associate a unique type to each variable throughout program executions, while we use a flow- and context-sensitive analysis. Following classic abstract interpreters [5], our analysis is performed by structural induction on the syntax, starting from a main entry point. It is thus unable to analyze functions in isolation. While this kind of modularity is a highlight of typing algorithms, we believe that it is not well suited to Python. Consider, for instance, that a call to a function can alter the value, and so the type, of a global variable, which is difficult to express in a type system. Moreover, even a simple function, such as `def f(a, b): return a + b`, has an unpredictable effect as the `+` operator can be overloaded to an arbitrary method by the programmer. We view type analysis as an instance of abstract interpretation, one which is slightly more abstract than classic value analysis. This view is not novel: [8] reconstructs Hindley-Milner typing rules as an abstract interpretation of the concrete semantics of the lambda calculus. One benefit of this unified view is the possibility to incorporate, in future work, some amount of value analysis through reduction. For instance, our analysis currently considers, to be sound, that any division can raise a `ZeroDivError`, which could be ruled out by a simple integer analysis. Finally, the correctness proof of our analysis is derived through a soundness theorem linking the concrete and the abstract semantics, in classic abstract interpretation form, and not by subject reduction. Both our analysis and type systems are conservative, but we replace the motto “well-typed programs cannot go wrong” with a guaranteed over-approximation of the possible (correct and incorrect) behaviors of the program.

Contributions. Our contributions are as follow:

- We present a sound static type analysis by abstract interpretation that handles most of Python constructions. Compared to classic analyses targetting static languages, we believe the uniqueness and precision lies in the combination of domains, allowing the analyzer to soundly know both nominal and structural types of manipulated objects, as well as the raised exceptions.
- Our analysis is based on several abstract domains that are combined together in a reduced product. A first, non-relational domain tracks for each variable the set of its possible types. A second domain infers type equalities between variables, which can achieve a form of parametric polymorphism. A third domain analyzes containers (such as lists).
- We provide concretization functions for our domains as well as selected transfer functions (for the sake of concision). We use the concretization-only setting of abstract interpretation to define how our analysis is sound with respect to a concrete semantics based on [13]. Compared to previous work on dynamic languages, the formalization of the concretization functions parameterized by the recency abstraction is novel.

² In practice, we are nevertheless limited to programs that do not use `eval`.

- We have implemented our analysis modularly within the Mopsa framework [20], and showed that it can analyze real-world benchmarks with few false alarms and reasonable efficiency, using the non-relational analysis. Moreover, we have shown that it performs better than existing Python type analyses, and can handle soundly program traits that they cannot, including introspection and exception handling.
- In order for our analysis to scale, our analyzer is able to read stub files containing Pythonic type annotations. We reused the Typedshed project [37], containing annotations of the standard library. We analyze a small (3kLOC) real-life utility called PathPicker [43] using more than 12 modules from the standard library.

Limitations. The two current limitations to the scalability of our approach are the standard library support, and the interprocedural analysis. The standard library is huge, and while some parts are written in Python others are written in C. We currently leverage the type annotations from Typedshed to support the standard library. We support most, but currently not all these annotations, and we add new Typedshed stub files when needed. The interprocedural analysis is based on inlining, which is costly but precise. It is improved with a cache in Section 6.2, and a more efficient interprocedural analysis is left as future work. The smashing-based abstraction of containers, combined with no information on the length of those containers creates spurious alarms. For example, in order to be sound, each list access raises an invalid access exception `IndexError`. The language support is wide enough to scale to programs a few thousand lines long, but still not complete. The analyzer does not support recursive functions for now, but the implementation would not be technically difficult (as loops and dynamic allocation are supported, respectively using a usual accelerated fixpoint computation – see Fig. 6 in [13] – and a recency abstraction [3]). It however detects recursive calls and stops at that point. Recursion is not used a lot in Python. In particular, there is no tail-call optimization, and the default recursion stack has depth 1000. Similarly, metaclasses are not supported but should not be technically difficult to implement. The `super` class and arbitrary code evaluation using `eval` are not supported. The latter feature is less used than in JavaScript (we never encountered it in our benchmarks), and solutions exist [17]. We could reuse [13] to handle generators. Asynchronous operators are not supported either [32].

Outline. Section 2 starts by recalling and slightly improving on the formalization of Python semantics from Fromherz et al. [13]. Section 3 then presents a non-relational static type analysis. A relational type equality domain refines the previous analysis in Section 4. Section 5 presents the analysis of containers, with the example of lists. Section 6 discusses our implementation and presents experimental results. Section 7 discusses related works and Section 8 concludes.

2 Concrete Semantics of Python

We first discuss the concrete semantics of Python, as our static analysis is stated as a sound abstract interpretation of this semantics. This semantics is the implementation of Python’s official interpreter, CPython (we focus on Python 3.7). It is thus not standardized nor formally defined. We rely on the reference manual and use CPython’s behavior and source code as an oracle in case of doubt. We use a slight evolution of the semantics proposed in [13], where we have adapted the semantics to get closer to the real Python language: while builtin objects, such as integers, were considered as primitive values, they are now objects allocated on the heap, which allows the creation of classes inheriting from builtin classes. We start by defining the memory state on which a Python program acts, and then describe a few parts of the concrete semantics of Python.

$$\begin{array}{l|l}
\mathbf{Id} \subseteq \mathit{string} & \mathbf{Addr} \stackrel{\text{def}}{=} \mathbf{Location} \times \mathbb{N} \\
\mathcal{E} \stackrel{\text{def}}{=} \mathbf{Id} \rightarrow \mathbf{Addr} \cup \mathbf{LocalErr} & \mathbf{ObjN} \stackrel{\text{def}}{=} \mathit{int}(a \in \mathbf{Addr}, i \in \mathbb{Z}) \cup \mathit{bool}(b) \\
\mathcal{H} \stackrel{\text{def}}{=} \mathbf{Addr} \rightarrow \mathbf{ObjN} \times \mathbf{ObjS} & \cup \mathit{string}(a, s \in \mathit{string}) \cup \mathbf{None} \cup \mathbf{NotImpl} \\
\mathcal{F} \stackrel{\text{def}}{=} \{ \mathit{cur}, \mathit{ret}, \mathit{brk}, & \cup \mathbf{List}(a, ls), ls \in \mathbf{Addr}^* \cup \mathbf{Method}(a, f) \\
\mathit{cont}, \mathit{exn} \ a, a \in \mathbf{Addr} \} & \cup \mathbf{Fun}(f) \cup \mathbf{Class}(c) \cup \mathbf{Instance}(a \in \mathbf{Addr}) \\
& \mathbf{ObjS} \stackrel{\text{def}}{=} \mathit{string} \rightarrow \mathbf{Addr}
\end{array}$$

■ **Figure 2** Concrete semantic domains.

2.1 Concrete Semantic Domain

The memory state consists in two parts: the environment \mathcal{E} and the heap \mathcal{H} , defined in Fig. 2. The environment \mathcal{E} is a partial, finite map from variable identifiers \mathbf{Id} to addresses \mathbf{Addr} . The set \mathbf{Addr} of heap addresses is infinite. To simplify the definition of our semantics and its abstraction, we assume that, up to isomorphism, \mathbf{Addr} has the form $\mathbf{Location} \times \mathbb{N}$ where $\mathbf{Location}$ is the set of program locations (in the following, a line number). We use the notation $@(l, n) \in \mathbf{Addr}$ to designate unambiguously the n -th address allocated at program location l . Due to scoping rules in Python, local variables may be locally undefined. We denote this using an additional value for local variables, called $\mathbf{LocalErr}$, as using such undefined variables results in an $\mathbf{UnboundLocalError}$ exception. The heap \mathcal{H} maps addresses to objects defined by their nominal and structural information. The nominal part gives the class and the value of the object, while the structural one gives for each attribute its allocation address.

Although everything is object in Python, we distinguish builtin objects from the other ones. Primitive objects include integers, strings, booleans, \mathbf{None} , $\mathbf{NotImpl}$ and lists (storing the addresses of each of its elements). Other containers such as dictionaries, sets, and tuples are handled similarly. These builtin objects are kept separate from the user-defined classes as their implementation uses low-level fields only accessible by CPython, which are hidden by the semantics. Some builtin objects (integers, strings, and lists only) depend on an address, being the class from which they are instantiated. This allows to handle classes inheriting from builtins: for example, if \mathbf{A} inherits from the \mathbf{int} class, $\mathbf{A}(10)$ is represented as $\mathbf{int}(@, 10)$, where $@$ points to the class of \mathbf{A} . This behavior was not expressible in the previous concrete semantics [13]. In the following, the notation $\mathbf{int}(10)$ denotes $\mathbf{int}(@, 10)$, where $@$ points to the class of integers (which happen in most cases), and similarly for other builtin objects. Methods bind an instance address and a function. Instances are defined by the address of the class from which they are instantiated. Classes and functions are also objects (we do not detail their inner structure). Concerning the structural part, a finite number of attributes may be added to classes, functions, and instances, so we additionally keep a map from attribute names to addresses for those objects.

Environment and Heap Example. Consider the code of Fig. 3. We start from an initial, empty state (\emptyset, \emptyset) , and show how the state of the environment e and heap h are *extended* after each step in Fig. 4. After the class declaration at lines 1 to 5, the identifier \mathbf{A} refers to the eponymous class, which is allocated at line 1, and is the first allocated object there, hence it has the address $@(1, 0)$. The instance of \mathbf{A} created at line 6 has an attribute \mathbf{val} being

```

1  class A:
2      def __init__(self):
3          self.update(0)
4      def update(self, x):
5          self.val = x * 2
6      x = A()
7      y = x.val
8      z = x
9      z.update('a')
10     if *: x.atr= 'b'

```

■ **Figure 3** Mutating objects: an example.

Line	$e \in \mathcal{E}$	$h \in \mathcal{H}$
5	$A \mapsto @(1, 0)$	$@(1, 0) \mapsto (\mathbf{Class} \ A, \{ __init__ \mapsto @(2, 0), \text{update} \mapsto @(4, 0) \})$ $@(2, 0) \mapsto (\mathbf{Fun}(__init__, \emptyset), @ (4, 0) \mapsto (\mathbf{Fun}(__update__, \emptyset))$
6	$x \mapsto @(6, 0)$	$@(6, 0) \mapsto (\mathbf{Instance} \ @(1, 0), \text{val} \mapsto @(5, 0))$ $@(5, 0) \mapsto (\mathbf{int} \ 0, \emptyset)$
7	$y \mapsto @(5, 0)$	
8	$z \mapsto @(6, 0)$	
9		$@(6, 0) \mapsto (\mathbf{Instance} \ @(1, 0), \text{val} \mapsto @ (5, 1))$ $@(5, 1) \mapsto (\mathbf{string} \ 'aa', \emptyset)$
10		Two possible heaps: either the heap from line 9, or: $@(6, 0) \mapsto (\mathbf{Instance} \ @(1, 0), \text{val} \mapsto @ (5, 1), \text{atr} \mapsto @ (10, 0))$ $@(10, 0) \mapsto (\mathbf{string} \ 'b', \emptyset)$

■ **Figure 4** Mutating objects example: state evolution.

the integer 0; it is created during the initialization of the class. After line 8, y maps to the integer stored in the field x of the instance of A . As z points to the same instance of A as x , we only need to add a binding between z and the instance's address. At line 9, the instance is updated using a method call, and the field val now maps to the string $'aa'$. After line 10, we have two possible heaps, depending on the non-deterministic choice $*$: either the previous heap, or a heap extended to add atr to the instance of A .

Flow tokens. Following [13], we present our semantic as a function from a set of states in precondition to a set of states in postcondition by induction on the program syntax. To provide a semantics for operations that do not return immediately, such as **raise**, we use continuations: we label states using *flow tokens* (elements of \mathcal{F}), so the states we consider are in $\mathcal{P}(\mathcal{F} \times \mathcal{E} \times \mathcal{H})$. Flow token *cur* represents the current flow on which all instructions that do not disrupt the control flow operate (e.g., assignments, but not **raise** or **return**). *ret* collects the set of states given by a **return** statement, while *brk*, *cont*, *exn* perform similar collections for the **break**, **continue**, **raise** statements, respectively. Each exception will be kept in a separate flow, so *exn* is indexed by the address of the exception object.

2.2 Semantics of Expressions and Statements

We denote by $\mathbb{E}[e]$ the semantics of expression e . This semantics has the following signature: $\mathbb{E}[e] : \mathcal{F} \times \mathcal{E} \times \mathcal{H} \rightarrow \mathcal{F} \times \mathcal{E} \times \mathcal{H} \times (\mathbf{Addr} \cup \{\perp\})$, so $\mathbb{E}[e]$ returns the address where the object associated with e is stored in the heap (or \perp if an exception is raised and no address is returned). The semantics of statements is written $\mathbb{S}[s]$ and has signature: $\mathcal{F} \times \mathcal{E} \times \mathcal{H} \rightarrow \mathcal{F} \times \mathcal{E} \times \mathcal{H}$. We do not describe the whole semantics of Python: we cover a few cases that are of interest for the upcoming type analysis; some other cases are described in [13] (the addition operator, conditionals, while loops and generators).

$$\begin{aligned}
\mathbb{E}[id](f, e, h) &\stackrel{\text{def}}{=} \\
&\text{if } f \neq \text{cur} \text{ then } (f, e, h), \perp \text{ else} \\
&\text{if } id \notin \text{dom } e \text{ then } \mathbb{S}[\text{raise NameError}](f, e, h), \perp \text{ else} \\
&\text{if } e(id) = \text{LocalErr} \text{ then } \mathbb{S}[\text{raise UnboundLocalError}](f, e, h), \perp \text{ else} \\
&(f, e, h), e(id) \\
\mathbb{S}[id = e](f, e, h) &\stackrel{\text{def}}{=} \\
&\text{let } (f, e, h), @ = \mathbb{E}[e](f, e, h) \text{ in} \\
&\text{if } f \neq \text{cur} \text{ then } (f, e, h) \text{ else } (\text{cur}, e[id \mapsto @], h)
\end{aligned}$$

■ **Figure 5** Concrete semantics of variable evaluation and assignment.

Variable Evaluation and Assignment. We define the semantics of variable evaluation and assignment in Fig. 5. To evaluate a variable identifier given an input state, we first check that the flow token is *cur*: otherwise, the state is returned with \perp instead of an address. Then, two errors may happen: if the variable is not defined in the program at all, a `NameError` is raised. If the variable is not defined in the current scope but defined somewhere else in the program, an `UnboundLocalError` is raised. Otherwise, the variable is evaluated into an address, and both the state and the address are returned.

To assign e to id , Python first starts by evaluating e : if other flows are created – for example if an exception is raised during the evaluation – they are just returned, and the assignment takes place only in the current flow. As most of the time, we want to update only the current flow, we introduce the following notation “`letcur (f, e, h), @ = e1 in e2`” which unfolds into “`let (f, e, h), @ = e1 in if f ≠ cur then (f, e, h), @ else e2`”.

Semantics of Attribute Accesses. We show how to access attribute s of expression e in Fig. 6. This is a formalization of one of Python’s complex behaviors, shown here to illustrate the complexity of the language. $e.s$ dispatches the attribute access to a method being either `__getattr__` or `__getattribute__`. The first method call usually ends up being `object.__getattribute__`. It can be overloaded by any class, which is supported in our implementation, but not very common, so we describe only the semantics of `object.__getattribute__`.

To evaluate `object.__getattribute__(e, s)`, we start by evaluating both e and s , which return object addresses: respectively $@_e$ and $@_s$. If s is not a string, a type error is raised. Then, we search for s in the parents class³ of the allocated object $h(@_e)$ using the function `mrosearch` (which takes as input a class and a string, and returns a parent class of the input, or \perp if the search is unsuccessful). The function `type` returns the class from which the object given in argument has been instantiated. If no class is found, we search for the attribute in the object only, using the low-level `has_field` and `get_field` operators. `get_field` is a low-level version of attribute access: it searches for the field locally, in the object structure, but it will not recursively search in the object’s class (nor its parents). It takes as input an object and

³ Actually, we search for s in the MRO of $h(@_e)$. The MRO of a class c is a list of classes from which c inherits, starting from its closest parents, to its most ancient one (usually, `object`). Even if multiple inheritances induce a direct acyclic graph, the inheritance relationship is linearized (using the algorithm described in [4]).

```

 $\mathbb{E}[\text{object}.\_\_getattribute\_\_(e, s)](f, e, h) \stackrel{\text{def}}{=}
\text{letcur } (f, e, h), @_e = \mathbb{E}[e](f, e, h) \text{ in}
\text{letcur } (f, e, h), @_s = \mathbb{E}[s](f, e, h) \text{ in}
\text{if } \text{isinstance}(h(@_s), \text{str}) \text{ then}
\text{let } \text{string } s = h(@_s) \text{ in}
\text{let } c = \text{mrosearch}(\text{type}(h(@_e)), s) \text{ in}
\text{if } c \neq \perp \text{ then}
\text{let } cs = h(\text{get\_field}(c, s)) \text{ in}
\text{if } \text{has\_field}(cs, "\_\_get\_\_") \wedge \text{has\_field}(cs, "\_\_set\_\_") \text{ then}
\mathbb{E}[\text{get\_field}(cs, "\_\_get\_\_")(cs, h(@_e), \text{type}(h(@_e)))](f, e, h)
\text{else if } \text{has\_field}(h(@_e), s) \text{ then } \mathbb{E}[\text{get\_field}(h(@_e), s)](f, e, h)
\text{else } \mathbb{E}[\text{get\_field}(c, s)](f, e, h)
\text{else if } \text{has\_field}(h(@_e), s) \text{ then } \mathbb{E}[\text{get\_field}(h(@_e), s)](f, e, h)
\text{else } \mathbb{S}[\text{raise AttributeError}](f, e, h), \perp
\text{else } \mathbb{S}[\text{raise TypeError}](f, e, h), \perp$ 
```

■ **Figure 6** Concrete semantics of attribute access.

a string and returns the address of the field defined by its arguments. Similarly, `has_field` checks for the presence of a field at the object structure only (contrary to `hasattr`). If a class c is found, let us denote by cs the object corresponding to the access of s in c . If cs is a data descriptor (meaning it has fields `__get__` and `__set__`), the result is the evaluation of cs 's `__get__` method. Otherwise, we return the attribute at e 's level (if it exists), or at c 's level otherwise.

Semantics of Exceptions. Finally, we showcase the use of multiple flows by defining the semantics of exceptions in Fig. 7. To raise an expression e , we evaluate e and check that it is an instance of the `BaseException` class. In that case, we change the flow token to exn , parameterized by the evaluation of e , and return the environment and heap. If not, we raise a type error. Now, let us consider the exception-catching mechanism. It behaves as follow: `tbody` is evaluated; if no exception is raised the evaluation is finished. Otherwise, if an exception (stored at address $@_{raised}$) is raised during this evaluation, we evaluate `exn`: if it is an exception (i.e, if it inherits from `BaseException`), and if the raised exception is an instance of `exn`, we evaluate `texc`, where the variable v is now bound to the raised exception (until the end of the evaluation of `texc`). Otherwise, we let the exception escape this control block.

3 A Non-relational Static Type Analysis

We present a non-relational type abstraction of Python semantics. The abstraction can infer the nominal and structural types of Python values in a flow-sensitive way, i.e., the types of a variable can vary from one program location to another. It is context-sensitive, supports object mutability and aliasing. For concision, we limit here the presentation to atomic types, such as numbers, strings and instances of user-defined classes. This abstraction will be completed with a type relation analysis in Section 4, and extended to containers in Section 5.

```

 $\mathbb{S}[\text{raise } e](f, e, h) =$ 
  letcur  $(f, e, h, @) = \mathbb{E}[e()](f, e, h)$  in
  if  $\text{isinstance}(h(@), \text{BaseException})$  then  $(\text{exn } @, e, h)$ 
  else  $\mathbb{S}[\text{raise TypeError}](f, e, h)$ 
 $\mathbb{S}[\text{try : tbody except exn as v : texc}](f, e, h) =$ 
  let  $(f, e, h) = \mathbb{S}[\text{tbody}](f, e, h)$  in
  if  $f \neq \text{exn\_}$  then  $(f, e, h)$ 
  else let  $\text{exn } @_{\text{raised}} = f$  in
    letcur  $(f, e, h), @_{\text{exn}} = \mathbb{E}[\text{exn}](\text{cur}, e, h)$  in
    if  $\neg \text{issubclass}(h(@_{\text{exn}}), \text{BaseException})$  then
       $\mathbb{S}[\text{raise TypeError}](\text{cur}, e, h)$ 
    else if  $\text{isinstance}(h(@_{\text{raised}}), h(@_{\text{exn}}))$  then
      let  $(f, e, h) = \mathbb{S}[\text{texc}](\text{cur}, e[v \mapsto @_{\text{raised}}], h)$  in  $(f, e \setminus \{v\}, h)$ 
    else  $(f, e, h)$ 

```

■ **Figure 7** Concrete semantics of exceptions.

3.1 Abstract Domain

The structure of the abstract domain \mathcal{D}^\sharp is defined in Fig. 8, along with the concretization functions, giving concrete meaning to the abstract states in Fig. 9. It is decomposed into three parts: a recency abstraction of allocated addresses, an environment abstraction, and a heap abstraction, all explained below. The example from the concrete semantics section is revisited in Sec. 3.2.

Recency Abstraction. To over-approximate a set of concrete addresses, we use a recency abstraction, as introduced in [3]. This abstraction maintains two kinds of information about allocated addresses. Firstly, the allocation site is preserved in order to distinguish between allocations at different program locations. Secondly, allocations at a same location $l \in \mathbf{Location}$ are partitioned into two abstract addresses via a recency criterion: *the* most recent allocation is represented with the abstract address $@^\sharp(l, \mathbf{r})$, while *all* the previous ones are abstracted by a unique abstract address $@^\sharp(l, \mathbf{o})$. The concretization function γ_{recency} takes as input the set of abstract addresses currently defined ($\rho \in \mathcal{P}(\mathbf{Addr}^\sharp)$), and yields a set of address abstraction functions, giving concrete meaning to abstract addresses, satisfying the conditions mentioned above. As addresses play an important part in every part of the abstract states, the other concretization operators are parameterized by an address abstraction $\alpha_{\mathbf{Addr}}$; we denote this parameterization with the following notation: $\gamma[\alpha_{\mathbf{Addr}}]$.

The allocation of a new abstract address is handled by the auxiliary function `alloc_addr`:

```

 $\mathbb{E}^\sharp[\text{alloc\_addr}(l)](\varphi, \rho \in \mathcal{P}(\mathbf{Addr}^\sharp), \epsilon, \eta) =$ 
  if  $@^\sharp(l, \mathbf{r}) \in \rho$  then  $@^\sharp(l, \mathbf{r}), \mathbb{S}^\sharp[@^\sharp(l, \mathbf{o}) \stackrel{\text{weak}}{=} @^\sharp(l, \mathbf{r})](\varphi, \rho, \epsilon, \eta)$ 
  else  $@^\sharp(l, \mathbf{r}), (\varphi, \rho \cup \{ @^\sharp(l, \mathbf{r}) \}, \epsilon, \eta)$ 

```

$\mathbf{Addr}^\# \stackrel{\text{def}}{=} \mathbf{Location} \times \{ \mathbf{r}, \mathbf{o} \}$ $\mathcal{F}^\# \stackrel{\text{def}}{=} \{ \mathit{cur}, \mathit{ret}, \mathit{brk}, \mathit{cont}, \mathit{exn} \ @^\#, \\ \quad \quad \quad @^\# \in \mathbf{Addr}^\# \}$ $\mathcal{E}^\# \stackrel{\text{def}}{=} \mathbf{Id} \rightarrow \mathcal{P}(\mathbf{Addr}^\# \cup \{ \mathbf{LocalErr} \})$ $\mathcal{H}^\# \stackrel{\text{def}}{=} \mathbf{Addr}^\# \rightarrow \mathcal{P}(\mathbf{ObjN}^\# \times \mathbf{ObjS}^\#)$ $\mathcal{D}^\# \stackrel{\text{def}}{=} \mathcal{F}^\# \rightarrow (\mathcal{P}(\mathbf{Addr}^\#) \times \mathcal{E}^\# \times \mathcal{H}^\#)$	$\mathbf{ObjN}^\# \stackrel{\text{def}}{=} \mathbf{AInt}(a) \cup \mathbf{AStr}(a) \\ \cup \mathbf{AMethod}(a, f) \\ \cup \mathbf{AClass}(c) \cup \mathbf{AFun}(f) \\ \cup \mathbf{AInst}(a), a \in \mathbf{Addr}^\#$ $\mathbf{ObjS}^\# \stackrel{\text{def}}{=} \{ \top \} \cup \\ (\mathcal{P}(\mathit{string}) \times (\mathit{string} \rightarrow \mathbf{Addr}^\#))$
---	--

■ **Figure 8** Definition of abstract states.

$$\begin{aligned} \gamma_{\text{recency}}(\rho \in \mathcal{P}(\mathbf{Addr}^\#)) &= \{ \alpha_{\mathbf{Addr}} : \mathbf{Addr} \rightarrow \mathbf{Addr}^\# \mid \\ &\quad (@^\#(l, o) \in \rho \implies \exists m \in \mathbb{N}, \forall i \leq m, \alpha_{\mathbf{Addr}}(@^\#(l, i)) = @^\#(l, o)) \wedge \\ &\quad (@^\#(l, r) \in \rho \implies \exists n \in \mathbb{N}, (\alpha_{\mathbf{Addr}}^{-1}(@^\#(l, r)) = \{ @^\#(l, n) \} \\ &\quad \quad \quad \wedge n = 1 + \max\{ i \mid \alpha_{\mathbf{Addr}}(@^\#(l, i)) = @^\#(l, o) \}) \} \end{aligned}$$

$$\begin{aligned} \gamma_{\mathcal{E}}[\alpha_{\mathbf{Addr}}](\epsilon \in \mathcal{E}^\#) &= \{ e \in \mathcal{E} \mid \forall v \in \text{dom } \epsilon, \alpha_{\mathbf{Addr}}(e(v)) \in \epsilon(v) \\ &\quad \vee \mathbf{LocalErr} \in \epsilon(v) \implies e(\mathit{id}) = \mathbf{LocalErr} \} \end{aligned}$$

$$\begin{aligned} \gamma_{\mathbf{ObjN}}[\alpha_{\mathbf{Addr}}](\mathbf{AInt}(@^\#)) &= \{ \mathit{int}(@, i) \mid i \in \mathbb{Z}, \alpha_{\mathbf{Addr}}(@) = @^\# \} \\ \gamma_{\mathbf{ObjN}}[\alpha_{\mathbf{Addr}}](\mathbf{AStr}(@^\#)) &= \{ \mathit{string}(@, s) \mid s \in \mathcal{P}(\mathit{str}), \alpha_{\mathbf{Addr}}(@) = @^\# \} \\ \gamma_{\mathbf{ObjN}}[\alpha_{\mathbf{Addr}}](\mathbf{AMethod}(@^\#, f)) &= \{ \mathbf{Method}(@, f) \mid \alpha_{\mathbf{Addr}}(@) = @^\# \} \\ \gamma_{\mathbf{ObjN}}[\alpha_{\mathbf{Addr}}](\mathbf{AClass}(c)) &= \{ \mathbf{Class}(c) \} \\ \gamma_{\mathbf{ObjN}}[\alpha_{\mathbf{Addr}}](\mathbf{AFun}(f)) &= \{ \mathbf{Fun}(f) \} \\ \gamma_{\mathbf{ObjN}}[\alpha_{\mathbf{Addr}}](\mathbf{AInst}(@^\#)) &= \{ \mathbf{Instance}(@) \mid \alpha_{\mathbf{Addr}}(@) = @^\# \} \end{aligned}$$

$$\begin{aligned} \gamma_{\mathbf{ObjS}}[\alpha_{\mathbf{Addr}}](\mu \in \mathcal{P}(\mathit{string}), f^\# \in \mathit{string} \rightarrow \mathbf{Addr}^\#) &= \{ f \in \mathbf{ObjS} = \mathit{string} \rightarrow \mathbf{Addr} \mid \\ &\quad \mu \subseteq \text{dom } f \subseteq \text{dom } f^\# \wedge \forall s \in \text{dom } f, \alpha_{\mathbf{Addr}}(f(s)) = f^\#(s) \} \\ \gamma_{\mathbf{ObjS}}[\alpha_{\mathbf{Addr}}](\top) &= \mathbf{ObjS} \end{aligned}$$

$$\begin{aligned} \gamma_{\mathcal{H}}[\alpha_{\mathbf{Addr}}](\eta \in \mathcal{H}^\#) &= \{ h \in \mathcal{H} \mid \forall @ \in \text{dom } \alpha_{\mathbf{Addr}}, h(@) = (n, s) \\ &\quad \wedge (\exists (n^\#, s^\#) \in \eta(\alpha_{\mathbf{Addr}}(@)), n \in \gamma_{\mathbf{ObjN}}[\alpha_{\mathbf{Addr}}](n^\#) \wedge s \in \gamma_{\mathbf{ObjS}}[\alpha_{\mathbf{Addr}}](s^\#)) \} \end{aligned}$$

$$\begin{aligned} \gamma_{\mathcal{F}}[\alpha_{\mathbf{Addr}}](\mathit{exn} @^\#) &= \{ \mathit{exn} @ \mid \alpha_{\mathbf{Addr}}(@) = @^\# \} \\ \gamma_{\mathcal{F}}[\alpha_{\mathbf{Addr}}](f \in \mathcal{F}^\#, f \neq \mathit{exn} _) &= \{ f \} \end{aligned}$$

$$\begin{aligned} \gamma(\varphi \in \mathcal{F}^\#, (\rho \in \mathcal{P}(\mathbf{Addr}^\#), \epsilon \in \mathcal{E}^\#, \eta \in \mathcal{H}^\#)) &= \{ (f, e, h) \in \mathcal{F} \times \mathcal{E} \times \mathcal{H} \mid \\ &\quad \alpha_{\mathbf{Addr}} \in \gamma_{\text{recency}}(\rho) \wedge f \in \gamma_{\mathcal{F}}[\alpha_{\mathbf{Addr}}](\varphi) \wedge e \in \gamma_{\mathcal{E}}[\alpha_{\mathbf{Addr}}](\epsilon) \wedge h \in \gamma_{\mathcal{H}}[\alpha_{\mathbf{Addr}}](\eta) \} \end{aligned}$$

$$\gamma_{\mathcal{D}}(\delta \in \mathcal{D}^\#) = \bigcup_{\varphi \in \text{dom } \delta} \gamma(\varphi, \delta(\varphi))$$

■ **Figure 9** Concretization of the abstract states.

17:12 Static Type Analysis by Abstract Interpretation of Python Programs

The semantics of `alloc_addr` is as follows. Given an allocation site $l \in \mathbf{Location}$, the function searches for the most recent allocation at the same location. If such an address $@^\sharp(l, \mathbf{r})$ exists, it should be moved to the pool of old addresses by copying its contents to the address $@^\sharp(l, \mathbf{o})$, which is done using a weak update $\mathbf{S}^\sharp \llbracket @^\sharp(l, \mathbf{o}) \stackrel{\text{weak}}{=} @^\sharp(l, \mathbf{r}) \rrbracket$. Otherwise, the state is extended with the new address $@^\sharp(l, \mathbf{r})$. In both cases, the newly allocated abstract address is $@^\sharp(l, \mathbf{r})$.

Environment Abstraction. The domain of abstract environments \mathcal{E}^\sharp maintains a non-relational map binding identifiers to a set of abstract addresses, with concretization $\gamma_{\mathcal{E}}[\alpha_{\mathbf{Addr}}]$. To support Python scoping, variables can also point to `LocalErr` to represent variables that can be locally undefined.

Heap Abstraction. The domain of abstract heaps \mathcal{H}^\sharp provides the Python objects associated to the addresses of the environment. Python objects are approximated by a nominal type abstraction \mathbf{ObjN}^\sharp and a structural type abstraction \mathbf{ObjS}^\sharp .

The nominal part keeps only the class information of the object and forgets about its value. The concretization $\gamma_{\mathbf{ObjN}}[\alpha_{\mathbf{Addr}}]$ maps abstract addresses to concrete ones. In particular, abstract instances of built-in values (integers, strings, booleans, `None`, `NotImpl`) are concretized into the set of corresponding built-in values (along with the class from which they were instantiated, for strings and integers, to support inheriting from these builtin classes). For instance: $\gamma_{\mathbf{ObjN}}[\alpha_{\mathbf{Addr}}](\mathbf{AInt}(@_{\mathbf{Class} \text{ int}}^\sharp)) = \{\mathbf{int}(@_{\mathbf{Class} \text{ int}}, i) \mid i \in \mathbb{Z}\}$, where the addresses subscripted by `Class int` represent the address of the built-in integer class. Similarly to the concrete, `AInt` implicitly means $\mathbf{AInt}(@_{\mathbf{Class} \text{ int}}^\sharp)$. `None` and `NotImpl` are abstracted as instances of their respective classes. The body of functions, methods and classes is not abstracted.

The structural part stores a map over-approximating the addresses referenced by the attributes of the object. Attributes may be added in some execution traces and not in others, hence, the map actually maintains the set of attributes that may exist at a given program point for all possible executions. We complement this map with a finite set under-approximating the set of attributes that are definitely present. This information is important to avoid raising spurious `AttributeError` exceptions for attributes that are definitely present. These properties are formally defined by the concretization $\gamma_{\mathbf{ObjS}}[\alpha_{\mathbf{Addr}}]$. The structural type abstraction may also be approximated as \top by the widening, in order to avoid having an infinite number of attributes being added to an instance.

Then, the concretization $\gamma_{\mathcal{H}}[\alpha_{\mathbf{Addr}}]$ of an abstract heap η is the set of heaps h such that each address $@$ is bound to an object (n, s) where: n is a concretization of n^\sharp , and s a concretization of s^\sharp , and the abstract object (n^\sharp, s^\sharp) is in the abstract heap at the abstract address $\alpha_{\mathbf{Addr}}(@)$. Note that the domain of the heap is defined by the recency abstraction, i.e. $\text{dom } \eta = \rho$.

Full State Abstraction. Flow tokens are also abstracted: only the exception token `exn` changes between the concrete and the abstract, to store an abstract address instead of a concrete one. This is formally described in the concretization $\gamma_{\mathcal{F}}[\alpha_{\mathbf{Addr}}]$. A whole abstract state maps flow tokens to abstract environments and heaps. To concretize a whole abstract state $\delta \in \mathcal{D}^\sharp$ using $\gamma_{\mathcal{D}}$, we concretize each image of δ separately and join the resulting concrete states. To concretize an element $\delta(\varphi) = (\rho, \epsilon, \eta)$ using γ , we first fix an address abstraction $\alpha_{\mathbf{Addr}}$ using the concretization of the recency abstraction ρ . Then, each part (flow token φ , environment ϵ , heap η) is concretized separately.

Line	$\epsilon \in \mathcal{E}^\#$	$\eta \in \mathcal{H}^\#$
5	$A \mapsto \{ @^\#(1, r) \}$	$@^\#(1, r) \mapsto \{ \mathbf{AClass} A, \{ __init__, update \}, __init__ \mapsto @^\#(2, r) \wedge update \mapsto @^\#(4, r) \};$ $@^\#(2, r) \mapsto \{ \mathbf{AFun}(__init__), \emptyset, \emptyset \}; @^\#(4, r) \mapsto \{ \mathbf{AFun}(update), \emptyset, \emptyset \}$
6	$x \mapsto \{ @^\#(6, r) \}$	$@^\#(6, r) \mapsto \{ \mathbf{AInst} @^\#(1, r), \{ val \}, val \mapsto @^\#(5, r) \}$ $@^\#(5, r) \mapsto \{ \mathbf{AInt}, \emptyset, \emptyset \}$
7	$y \mapsto \{ @^\#(5, r) \}$	
8	$z \mapsto \{ @^\#(6, r) \}$	
9	$y \mapsto \{ @^\#(5, o) \}$	$@^\#(5, r) \mapsto \{ \mathbf{AStr}, \emptyset, \emptyset \}$ $@^\#(5, o) \mapsto \{ \mathbf{AInt}, \emptyset, \emptyset \}$
10		$@^\#(6, r) \mapsto \{ \mathbf{AInst} @^\#(1, r), \{ val \}, val \mapsto @^\#(5, r) \wedge atr \mapsto @^\#(10, r) \}$ $@^\#(10, r) \mapsto \{ \mathbf{AStr}, \emptyset, \emptyset \}$

■ **Figure 10** Evolution of the abstract states of the example from Fig. 3.

3.2 Example

To illustrate our abstraction, let us consider the example shown previously in Fig. 3. We summarize the evolution of the abstract state in Fig. 10, for the current flow *cur*. After the declaration of the class **A**, the variable **x** is assigned the address $@^\#(6, r)$, representing the instance of **A** allocated at line 6 and having a unique attribute **val**. This attribute points to the address $@^\#(5, r)$, representing the integer result of the multiplication at line 5. After assigning the addresses $@^\#(5, r)$ and $@^\#(6, r)$ to **y** and **z** respectively (which changes the environment only), the call to **z.update('a')** leads to two changes. Firstly, during the evaluation of **x * 2** at line 5, a new address is allocated for the resulting string. Since the address $@^\#(5, r)$ already exists, it is renamed to $@^\#(5, o)$ to denote that it is no longer the most recent allocation. Consequently, the variable **y** now points to $@^\#(5, o)$ and the object pointed by this address remains an integer **AInt**. The second change affects the heap to ensure that the most recent allocation $@^\#(5, r)$ points now to a string object **AStr**. After line 10, a new string is allocated and assigned to the attribute **atr** belonging to the address $@^\#(6, r)$. Since this change is performed in only one branch of the **if** statement, **atr** is not added to the under-approximation of attributes.

We illustrate our concretization by linking the final abstract state, at line 10 in Fig. 10, to the concrete one in Fig. 4. In the abstract, $\rho = \{ @^\#(1, r), @^\#(5, o), @^\#(5, r), @^\#(6, r), @^\#(10, r) \}$. Let us define $\alpha_{\mathbf{Addr}}^{\text{ex}} = @^\#(1, 0) \mapsto @^\#(1, r); @^\#(6, 0) \mapsto @^\#(6, r); @^\#(5, 0) \mapsto @^\#(5, o); @^\#(5, 1) \mapsto @^\#(5, r); @^\#(10, 0) \mapsto @^\#(10, r)$. We can check that $\alpha_{\mathbf{Addr}}^{\text{ex}}$ is one of the abstraction functions defined in $\gamma_{\text{recency}}(\rho)$. In addition, it is the abstraction function whose domain is the set of addresses defined in the concrete example (Fig. 4). The concretization of the environment is unambiguous as the abstract addresses always represent only one concrete address (adding another call to **update** in the example would mean that two concrete addresses would be mapped to $@^\#(5, o)$ at the end). Continuing the example, we get that $\gamma_{\mathbf{ObjN}}[\alpha_{\mathbf{Addr}}^{\text{ex}}](\mathbf{AInst}(@^\#(1, r))) = \{ \mathbf{Instance}@^\#(1, 0) \}$. The structural type concretization concerning the attributes of the instance of **A** yields two different cases: $\gamma_{\mathbf{ObjS}}[\alpha_{\mathbf{Addr}}^{\text{ex}}](\{ val \}, val \mapsto @^\#(5, r) \wedge atr \mapsto @^\#(10, r)) = \{ f_1, f_2 \}$, with $f_1 = val \mapsto @^\#(5, 1)$, and $f_2 = f_1[at \mapsto @^\#(10, 0)]$ (depending on the addition of **atr** to the instance). The concrete heaps mentioned in Fig. 4 are part of the concretization of the abstract heap.

3.3 Abstract Transfer Functions

The abstract evaluation of expressions and statements is very close to the concrete one. We show in Fig. 11 the transfer functions of the assignment, object instantiation, and attribute addition. The signature of the abstract evaluation $\mathbb{E}^\# \llbracket e \rrbracket$ is $(\mathcal{F}^\# \times \mathcal{P}(\mathbf{Addr}^\#) \times \mathcal{E}^\# \times \mathcal{H}^\#) \rightarrow (\mathcal{F}^\# \times \mathcal{P}(\mathbf{Addr}^\#) \times \mathcal{E}^\# \times \mathcal{H}^\#) \times (\mathbf{Addr}^\# \cup \{\perp\})$, so $\mathbb{E}^\# \llbracket e \rrbracket$ returns the abstract address where the object associated with e is stored, along with the updated abstract state. The semantics of statements has a similar signature except that it only returns the updated abstract state. Similarly to the lift from γ to $\gamma_{\mathcal{D}}$, both semantics can be implicitly lifted to $\mathcal{D}^\#$, in the case of disjunctive evaluations or of multiple flow tokens (when an expression is evaluated into different types, or nondeterministically raises an exception).⁴

To perform an assignment $\mathbf{x} = \mathbf{e}$, we evaluate \mathbf{e} in the abstract, and change the abstract environment ϵ accordingly. The evaluation of \mathbf{e} may be disjunctive (if the expression may evaluate in multiple abstract addresses, or raises an exception in some cases) and in this case, states are merged by their flow tokens.

`object.__new__` is the function used to instantiate most classes. To analyze this call, we evaluate \mathbf{e} . If it is a class, we call the recency abstraction to allocate an instance, and return the result of this evaluation, where the abstract heap η is extended with this new address. Otherwise, a type error is raised.

`object.__setattr__` is the function usually called for an attribute update: $\mathbf{x}.\mathbf{attr} = \mathbf{e}$. It is similar in its complexity and shape to the concrete attribute access (Fig. 6). The complexity is due to the notion of *data descriptors*, stored in a parent class of an instance: they can preempt attribute addition and process it as a call to their own `__set__` method. In most cases, however, the `set_field` function will be called. In this case, we take the evaluation of \mathbf{x} as an address $@_x^\#$; we then fetch the attribute abstraction for this address. We update the abstraction map and store it as f'_x . Then, if the address is recent, we know that it represents only one address in the concrete. Thus, the attribute will be always defined in the object, and we can add it to the underapproximation of the attributes. If the address is old, it may summarize multiple concrete addresses, and the attribute will only be modified in f_x by the execution of the assignment. Note that Python also supports attribute update through the `setattr` function. Contrary to the assignment $\mathbf{x}.\mathbf{attr} = \mathbf{e}$ where Python's syntax ensures that `attr` is a constant string, `setattr` can take into argument an arbitrary string, which would result in the structural abstraction of the targeted object to be put to top. In that case, we can enable a constant string abstraction to refine the abstract value of the attribute name and help regain precision.

Join Operator and Widening. Going back to the abstract state definition (Fig. 8), we notice that only $\mathbf{ObjS}^\#$ can be infinite. We thus define a widening operator lifting the structural type abstraction to \top if too many attributes are added. The set of addresses is finite due to the finite number of program locations. Joining two abstract states is done pointwise: by merging states having the same flow tokens, joining the sets for the recency abstraction and the maps for the abstract environment and for the abstract heap.

Analysis of Functions. The analysis of functions is performed in a context-sensitive fashion, by inlining: when a function call is reached, we substitute the call by the body of the function and analyze it. This scheme supports easily dynamic dispatch as well as calling anonymous functions defined using `lambda`.

⁴ i.e. $\mathbb{S}^\# \llbracket stmt \rrbracket (\delta \in \mathcal{D}^\#) = \cup_{\varphi \in \text{dom } \delta, (\rho, \epsilon, \eta) \in \delta(\varphi)} \mathbb{S}^\# \llbracket stmt \rrbracket (\varphi, \rho, \epsilon, \eta)$

$$\begin{aligned}
& \mathbb{S}^\# \llbracket x = e \rrbracket (\varphi, \rho, \epsilon, \eta) \stackrel{\text{def}}{=} \\
& \quad \text{letcur } (\varphi, \rho, \epsilon, \eta), @^\# = \mathbb{E}^\# \llbracket e \rrbracket (\varphi, \rho, \epsilon, \eta) \text{ in } \varphi, \rho, \epsilon[x \mapsto \{ @^\# \}], \eta \\
& \mathbb{E}^\# \llbracket \text{object}._ _ \text{new} _ _ (e)^{\text{loc}} \rrbracket (\varphi, \rho, \epsilon, \eta) \stackrel{\text{def}}{=} \\
& \quad \text{letcur } (\varphi, \rho, \epsilon, \eta), @_e^\# = \mathbb{E}^\# \llbracket e \rrbracket (\varphi, \rho, \epsilon, \eta) \text{ in} \\
& \quad \text{if } (\text{fst} \circ \eta)(@_e^\#) = \mathbf{AClass} \ c \ \text{then} \\
& \quad \quad \text{letcur } (\varphi, \rho, \epsilon, \eta), @^\# = \mathbb{E}^\# \llbracket \text{alloc_addr}(\text{loc}) \rrbracket (\varphi, \rho, \epsilon, \eta) \text{ in } (\varphi, \rho, \epsilon, \eta[@^\# \mapsto \emptyset]), @^\# \\
& \quad \text{else } \mathbb{S}^\# \llbracket \text{raise TypeError} \rrbracket (\varphi, \rho, \epsilon, \eta), \perp \\
& \mathbb{E}^\# \llbracket \text{object}._ _ \text{setattr} _ _ (x, \text{attr} \in \text{string}, e) \rrbracket (\varphi, \rho, \epsilon, \eta) \stackrel{\text{def}}{=} \\
& \quad \text{letcur } (\varphi, \rho, \epsilon, \eta), @_x^\# = \mathbb{E}^\# \llbracket x \rrbracket (\varphi, \rho, \epsilon, \eta) \text{ in} \\
& \quad \text{let } c = \text{mrosearch}^\#(\text{type}^\#(@_x^\#), \text{attr}) \text{ in} \\
& \quad \text{if } c \neq \perp \ \text{then let } f = \text{get_field}^\#(\text{type}^\#(@_x^\#), \text{attr}) \text{ in} \\
& \quad \quad \text{if } \text{has_field}^\#(f, \text{"_ _ get _ _"}) \wedge \text{has_field}^\#(f, \text{"_ _ set _ _"}) \ \text{then} \\
& \quad \quad \quad \mathbb{E}^\# \llbracket (\text{get_field}^\#(f, \text{"_ _ set _ _"}))(c, @_x^\#, e) \rrbracket \\
& \quad \quad \text{else } \mathbb{E}^\# \llbracket \text{set_field}^\#(@_x^\#, \text{attr}, e) \rrbracket (\varphi, \rho, \epsilon, \eta) \\
& \quad \text{else } \mathbb{E}^\# \llbracket \text{set_field}^\#(@_x^\#, \text{attr}, e) \rrbracket (\varphi, \rho, \epsilon, \eta) \\
& \mathbb{E}^\# \llbracket \text{set_field}^\#(@_x^\#, \text{attr}, e) \rrbracket (\varphi, \rho, \epsilon, \eta) \stackrel{\text{def}}{=} \\
& \quad \text{letcur } (\varphi, \epsilon, \eta), @_e^\# = \mathbb{E}^\# \llbracket e \rrbracket (\varphi, \rho, \epsilon, \eta) \text{ in let } (t_x, (u_x, f_x)) = \eta(@_x^\#) \text{ in} \\
& \quad \text{let } f'_x = f_x[\text{attr} \mapsto @_e^\#] \text{ in} \\
& \quad \text{if recent_addr } @_x^\# \ \text{then } (\varphi, \rho, \epsilon, \eta[@_x^\# \mapsto (t_x, (u_x \cup \{ \text{attr} \}, f'_x)])) \\
& \quad \text{else } (\varphi, \rho, \epsilon, \eta[@_x^\# \mapsto (t_x, (u_x, f'_x)]))
\end{aligned}$$

■ **Figure 11** Examples of abstract transfer functions.

► **Theorem 1.** *Our analysis is sound: the abstract states computed by our abstract transfer functions over-approximate the concrete states reachable during any program execution. More formally, for any Python statement s : $\forall \delta \in \mathcal{D}^\#, \mathbb{S} \llbracket s \rrbracket \circ \gamma_{\mathcal{D}}(\delta) \subseteq \gamma_{\mathcal{D}} \circ \mathbb{S}^\# \llbracket s \rrbracket(\delta)$*

This theorem is proved by mutual structural induction on the structure of Python statements and expressions. The proof is not detailed due to space constraints. The abstract transfer functions of statements and expressions are close to the concrete ones, which makes the proof simple. For example, the semantics of `object.__getattr__` is the same in the concrete and in the abstract, up to the low-level operators `get_field`, `has_field`, `type`, `isinstance`.

4 Relational Analysis using Parametric Polymorphism

The analysis presented in Sec. 3 is polymorphic, as a variable may be abstracted as a set of addresses of different types. However, bounded parametric polymorphism *à la ML* is impossible to express in this abstraction as we cannot infer that two variables pointing to multiple addresses have the same type. From an abstract interpretation point of view, we lack a relational domain.

Example. Consider the following program:

```

1  if *: x, y = 1, 2
2  else: x, y = 'a', 'b'
3  z = x + y
    
```

Our non-relational analysis can infer after line 2 that both `x` and `y` have type `int` or `str`. However, it cannot show that `x` and `y` are either both `int` or both `str`, and thus it raises a `false TypeError` alarm when evaluating `x + y`.

Type Equality Abstract Domain. We introduce an abstract domain $\mathcal{Q}^\# \stackrel{\text{def}}{=} \mathbf{Id} \rightarrow \mathbb{N}$ to track type equalities between variables. It is defined as a partitioning of program identifiers \mathbf{Id} into equivalence classes of equally typed variables. Given $\kappa \in \mathcal{Q}^\#$, we ensure that two variables `x` and `y` verifying $\kappa(x) = \kappa(y)$ will have the same nominal type. More precisely, we define an abstract equivalence relation $\equiv_\eta^\# \subseteq \mathbf{ObjN}^\# \times \mathbf{ObjN}^\#$ between nominal types:

$$\begin{aligned} \equiv_\eta^\# \stackrel{\text{def}}{=} & \{ (\mathbf{AInt}(@_1^\#), \mathbf{AInt}(@_2^\#)) \mid (\text{fst } \circ \eta)(@_1^\#) \equiv_\eta^\# (\text{fst } \circ \eta)(@_2^\#) \} \\ & \cup \{ (\mathbf{AStr}(@_1^\#), \mathbf{AStr}(@_2^\#)) \mid (\text{fst } \circ \eta)(@_1^\#) \equiv_\eta^\# (\text{fst } \circ \eta)(@_2^\#) \} \\ & \cup \{ (\mathbf{AFun} -, \mathbf{AFun} -) \} \cup \{ (\mathbf{AClass} c, \mathbf{AClass} c) \} \\ & \cup \{ (\mathbf{AMethod}(@_1^\#, -), \mathbf{AMethod}(@_2^\#, -)) \mid (\text{fst } \circ \eta)(@_1^\#) \equiv_\eta^\# (\text{fst } \circ \eta)(@_2^\#) \} \\ & \cup \{ (\mathbf{AInst}(@_1^\#), \mathbf{AInst}(@_2^\#)) \mid (\text{fst } \circ \eta)(@_1^\#) \equiv_\eta^\# (\text{fst } \circ \eta)(@_2^\#) \} \end{aligned}$$

The concretization function $\gamma_{\mathcal{Q}} \in \mathcal{Q}^\# \rightarrow \mathcal{P}(\mathcal{F} \times \mathcal{E} \times \mathcal{H})$ gives the set of concrete states verifying the equality constraints of an abstract element in $\mathcal{Q}^\#$:

$$\gamma_{\mathcal{Q}}(\kappa) \stackrel{\text{def}}{=} \{ (f, e, h) \mid \forall x, y \in \text{dom } \kappa : \kappa(x) = \kappa(y) \implies (\text{fst } \circ h \circ e)(x) \equiv_h (\text{fst } \circ h \circ e)(y) \}$$

where $\equiv_h \subseteq \mathbf{ObjN} \times \mathbf{ObjN}$ is the concrete equivalence relation between nominal types in $h \in \mathcal{H}$, derived from $\equiv_\eta^\#$ as:

$$\begin{aligned} n_1 \equiv_h n_2 \Leftrightarrow & \exists n_1^\#, n_2^\# \in \mathbf{ObjN}^\#, \exists \eta \in \mathcal{H}^\#, \exists \alpha_{\mathbf{Addr}} \in \gamma_{\text{recency}}(\text{dom } \eta) : n_1^\# \equiv_\eta^\# n_2^\# \wedge \\ & n_1 \in \gamma_{\mathbf{ObjN}}[\alpha_{\mathbf{Addr}}](n_1^\#) \wedge n_2 \in \gamma_{\mathbf{ObjN}}[\alpha_{\mathbf{Addr}}](n_2^\#) \wedge h \in \gamma_{\mathcal{H}}[\alpha_{\mathbf{Addr}}](\eta) \end{aligned}$$

Reduced Product. In order to perform a type analysis with bounded parametric polymorphism, we construct a reduced product $\mathcal{D}_P^\#$ of the equality domain $\mathcal{Q}^\#$ and the non-relational domains $\mathcal{E}^\#$ and $\mathcal{H}^\#$ of Sec. 3 as follows:

$$\begin{aligned} \mathcal{D}_P^\# \stackrel{\text{def}}{=} & \mathcal{F}^\# \rightarrow (\mathcal{P}(\mathbf{Addr}^\#) \times \mathcal{E}^\# \times \mathcal{H}^\# \times \mathcal{Q}^\#) \\ \gamma_P(\delta_p \in \mathcal{D}_P^\#) = & \bigcup_{\substack{\varphi \in \text{dom } \delta_p \\ \delta_p(\varphi) = (\rho, \epsilon, \eta, \kappa)}} \gamma(f, (\rho, \epsilon, \eta)) \cap \gamma_{\mathcal{Q}}(\kappa) \end{aligned}$$

Two reduction operators $\psi_\uparrow, \psi_\downarrow \in (\mathcal{P}(\mathbf{Addr}^\#) \times \mathcal{E}^\# \times \mathcal{H}^\# \times \mathcal{Q}^\#) \rightarrow (\mathcal{P}(\mathbf{Addr}^\#) \times \mathcal{E}^\# \times \mathcal{H}^\# \times \mathcal{Q}^\#)$ are proposed to refine product states by propagating information between domains (they are extended pointwise so that $\psi_\downarrow, \psi_\uparrow \in \mathcal{D}_P^\# \rightarrow \mathcal{D}_P^\#$):

1. The reduction ψ_\uparrow enriches κ with new type equalities. It searches for variables `x` and `y` such that both of them point to singleton objects with equivalent nominal types:

$$\begin{aligned} \epsilon(x) = \{ @_x^\# \} \quad \eta(@_x^\#) = \{ (n_x, _) \} \\ \epsilon(y) = \{ @_y^\# \} \quad \eta(@_y^\#) = \{ (n_y, _) \} \quad \wedge \quad n_x \equiv_\eta^\# n_y \end{aligned}$$

In such case, we add the type equality $\kappa(x) = \kappa(y)$.

Before reduction	After reduction
$\epsilon = x \mapsto @^\sharp(1, o) \wedge y \mapsto @^\sharp(1, r),$	ϵ
$\eta = @^\sharp(1, o) \mapsto \{\mathbf{AInt}, \emptyset, \emptyset\} \wedge @^\sharp(1, r) \mapsto \{\mathbf{AInt}, \emptyset, \emptyset\},$	η
$\kappa = \perp$	$\kappa = x \mapsto 0, y \mapsto 0$

■ **Figure 12** Example of ψ_\downarrow reduction.

2. The reduction operator ψ_\downarrow refines the non-relational heap η whenever two variables x and y are equally typed in κ and the type of x is more precise. We do so by pruning away the objects referenced by y that are not equivalent to any object pointed by x .

► **Theorem 2.** *The reduced product is sound, meaning that the reduction operators do not affect the global product concretization: $\forall \delta \in \mathcal{D}_P^\sharp, \gamma_P(\delta) = \gamma_P(\psi_\uparrow(\delta)) = \gamma_P(\psi_\downarrow(\delta))$*

Example. Let us consider again the previous motivating example. After the assignment $x, y = 1, 2$, both x and y point to singleton integer objects, which allows us to apply ψ_\uparrow in order to infer the type equality of x and y (the state is shown in Fig. 12). The same reasoning is applied after the assignment $x, y = 'a', 'b'$ in the **else** branch. Consequently, the equality is preserved after joining the two abstract states at line 3. When evaluating x in the addition expression, a disjunction with two cases is created, one for each referenced abstract object. In each case, the reduction operator ψ_\downarrow is applied to refine the type of y according to the type of x . Therefore, at the end of the program, we infer that no **TypeError** is raised. Moreover, the reduction ψ_\uparrow will find that x, y , and z have the same type.

Bounded Parametric Polymorphism. In the motivating example, our analysis morally infers that x, y, z have type $\alpha \in \{\mathbf{int}, \mathbf{str}\}$. We believe this is close to bounded parametric polymorphism. In future work, we want to combine relationality with partial function summaries to deduce that f has type $\alpha \rightarrow \alpha, \alpha \in \{\mathbf{int}, \mathbf{str}\}$ in the program below.

```

1  def f(x, y): return x + y
2  f(1, 2)
3  f('a', 'b')
```

5 Independent Container Abstractions

Containers are abstracted independently from the rest of the analysis. We show the example of a smashing abstraction [6] for lists. The analysis of dictionaries is implemented similarly: their keys and their values are smashed separately. We have also implemented an expansion-based analysis for tuples.

The smashing abstraction summarizes all the list elements into one “content” variable. Hence, we can infer whether an abstract address is a list, and we can moreover infer the type of list elements using the content variable. As the content variable can have arbitrary abstract values, the abstraction can represent heterogeneous as well as nested lists, which are supported in Python.

Abstract Domain. We add a new nominal type for lists in \mathbf{ObjN}^\sharp , denoted as $\mathbf{AList}(@^\sharp \in \mathbf{Addr}^\sharp)$, the address representing the class from which the list is instantiated, to handle classes inheriting from lists (\mathbf{AList} implicitly means $\mathbf{AList}(@^\sharp_{\mathbf{Class}} \text{list})$). We also extend the set of identifiers into \mathbf{Id}^+ , adding a new kind of identifiers, $\text{List } @^\sharp$, to denote content variables ($@^\sharp$ is the address of the list).

$$\begin{aligned}
 \mathbb{E}^\# \llbracket [e_1, \dots, e_n]^l \rrbracket (f, \rho, \epsilon, \eta) = & \\
 \text{letcur } (\varphi, \rho, \epsilon, \eta), @^\# = \mathbb{E}^\# \llbracket \text{alloc_addr}(l) \rrbracket (\varphi, \rho, \epsilon, \eta) \text{ in} & \\
 \text{letcur } (\varphi, \rho, \epsilon, \eta) = \mathbb{S}^\# \llbracket \text{List } @^\# = e_1 \rrbracket (\varphi, \rho, \epsilon, \eta) \text{ in} & \\
 \text{letcur } (\varphi, \rho, \epsilon, \eta) = \mathbb{S}^\# \llbracket \text{List } @^\# \stackrel{\text{weak}}{=} e_2 \rrbracket (\varphi, \rho, \epsilon, \eta) \text{ in} & \\
 \dots & \\
 \text{letcur } (\varphi, \rho, \epsilon, \eta) = \mathbb{S}^\# \llbracket \text{List } @^\# \stackrel{\text{weak}}{=} e_n \rrbracket (\varphi, \rho, \epsilon, \eta) \text{ in } (f, \rho, \epsilon, \eta), @^\# & \\
 \mathbb{E}^\# \llbracket \text{list.append}(l, e) \rrbracket (f, \rho, \epsilon, \eta) = & \\
 \text{letcur } (\varphi, \rho, \epsilon, \eta), @_l^\# = \mathbb{E}^\# \llbracket l \rrbracket (\varphi, \rho, \epsilon, \eta) \text{ in} & \\
 \text{letcur } (\varphi, \rho, \epsilon, \eta), @_e^\# = \mathbb{E}^\# \llbracket e \rrbracket (\varphi, \rho, \epsilon, \eta) \text{ in} & \\
 \text{if } \text{fst } \circ \eta(@_l^\#) = \mathbf{AList}(-) \text{ then} & \\
 \quad \mathbb{E}^\# \llbracket \text{None} \rrbracket \circ \mathbb{S}^\# \llbracket \text{List } @_l^\# \stackrel{\text{weak}}{=} e \rrbracket (\varphi, \rho, \epsilon, \eta) & \\
 \text{else } \mathbb{S}^\# \llbracket \text{raise TypeError} \rrbracket (\varphi, \rho, \epsilon, \eta), \perp & \\
 \\
 \gamma_{\text{lists}}(\varphi, \rho, \epsilon, \eta) = \{ (f, e, h) \mid \alpha_{\mathbf{Addr}} \in \gamma_{\text{recency}}(\rho) \wedge (f', e', h') \in \gamma[\alpha_{\mathbf{Addr}}](\varphi, (\rho, \epsilon, \eta)) & \\
 (\forall v \in \text{dom } e', e(v) = e'(v)) \wedge (\forall @ \in \text{dom } h', h(@) = h'(@)) \wedge \forall v \in \text{dom } \epsilon, & \\
 (@^\#(l, m) \in \epsilon(v) \wedge (\mathbf{AList}(@_c^\#, \emptyset, \emptyset) \in \eta(@^\#(l, m))) \implies (\alpha_{\mathbf{Addr}}(e(v)) = @^\#(l, m) & \\
 \wedge \exists n \in \mathbb{N}, h(e(v)) = (\mathbf{List}(@_c, (@_1, \dots, @_n), \emptyset)) \wedge & \\
 \alpha_{\mathbf{Addr}}(@_c) = @_c^\# \wedge \forall 1 \leq i \leq n, \alpha_{\mathbf{Addr}}(@_i) \in \epsilon(\text{List } @^\#(l, m)) \} &
 \end{aligned}$$

■ **Figure 13** List transfer functions & extended concretization.

Transfer Functions. Fig. 13 presents the abstract semantics of list allocation. We start by allocating the address of the list through the recency abstraction. Then, we assign, using weak updates, each element of the list to the content variable, and return the address of the list. Adding an element `e1` to the list `l` using the function `list.append(l, e1)` is also simple. First, we evaluate `l` into an object and check that it is a list. From the evaluation of `l`, we get the address location $@^\#(l, m)$, letting us access the content variable `List @#(l, m)`. Then, we perform a weak update with the element `e1`.

We emphasize that these transfer functions are independent from most of the analysis: they only need an abstract domain handling address allocation, and another handling assignments. It could for example be reused in the case of a value analysis, or with another allocation-site abstraction.

Concretization. We extend the concretization γ from Fig. 9 into a concretization γ_{lists} taking the list abstraction into account, presented in Fig. 13. To simplify the presentation, $\gamma_{\text{lists}}(\varphi, \rho, \epsilon, \eta)$ employs a variant of γ where the address abstraction $\alpha_{\mathbf{Addr}}$ is fixed, denoted as $\gamma[\alpha_{\mathbf{Addr}}]$. Given an address abstraction $\alpha_{\mathbf{Addr}} \in \gamma_{\text{recency}}(\rho)$, $\gamma[\alpha_{\mathbf{Addr}}]$ provides partially concretized states (f', e', h') that ignore content identifiers as well as the **AList** nominal type. We then extend the (e', h') states into concrete states (e, h) : identifiers v that may be lists are added to the environment, the list defined in v is allocated in the concrete heap h at address $e(v)$ with an arbitrary size, and its element addresses are constrained to match with content of the abstract environment of the content variable.

Example. Consider the program $\mathbf{l} = [\text{'a'}^{l_1}, \text{'b'}^{l_2}, \text{'c'}^{l_3}]$. We use labels l_i to denote the program location of each string (program location are actually line numbers and column ranges). In the current flow cur , we get the following abstract environment and heap:

$$\begin{aligned} \epsilon(l) &= \{ @^\#(1, \mathbf{r}) \} & \epsilon(\text{List } @^\#(1, \mathbf{r})) &= \{ @^\#(l_1, \mathbf{r}), @^\#(l_2, \mathbf{r}), @^\#(l_3, \mathbf{r}) \} \\ \eta(@^\#(1, \mathbf{r})) &= (\mathbf{AList}, \emptyset, \emptyset) & \eta(@^\#(l_i, \mathbf{r})) &= (\mathbf{AString}, \emptyset, \emptyset), 1 \leq i \leq 3 \end{aligned}$$

$\epsilon(l)$ is bound to the abstract list address, allocated at location 1 and being a recent address. The list variable $\text{List } @^\#(1, \mathbf{r})$ may now point to three strong addresses, each representing one of the strings.

Nested Lists. Our encoding also works for nested lists. In the case of two nested lists, the outermost list variable would point to the address of the innermost list abstract address. These two abstract addresses would also be different because their program locations are different. For example, $\mathbf{l} = [1^{l_1}, [2.3^{l_2}]^i]^o$ yields the following abstract state (l_1, l_2 are program locations for the numbers, while i, o are program locations from the inner and the outer list respectively). The variable corresponding to the outer list $\text{List } @^\#(o, \mathbf{r})$ maps to two addresses, including the one of the inner list $@^\#(i, \mathbf{r})$.

$$\begin{aligned} \epsilon(l) &= \{ @^\#(o, \mathbf{r}) \} & \eta(@^\#(o, \mathbf{r})) &= (\mathbf{AList}, \emptyset, \emptyset) \\ \epsilon(\text{List } @^\#(o, \mathbf{r})) &= \{ @^\#(l_1, \mathbf{r}), @^\#(i, \mathbf{r}) \} & \eta(@^\#(i, \mathbf{r})) &= (\mathbf{AList}, \emptyset, \emptyset) \\ \epsilon(\text{List } @^\#(i, \mathbf{r})) &= \{ @^\#(l_2, \mathbf{r}) \} & \eta(@^\#(l_1, \mathbf{r})) &= (\mathbf{AInt}, \emptyset, \emptyset) \\ & & \eta(@^\#(l_2, \mathbf{r})) &= (\mathbf{AFloat}, \emptyset, \emptyset) \end{aligned}$$

This also works for arbitrary nesting. For example, let us consider the following program:

```

1  x = 1
2  for i in range(10): x = [x]

```

With the usual accelerated fixpoint computation, we reach an overapproximation of the concrete state, where x is (an integer or) a nested list containing only integers, but we lose the nest level.

$$\begin{aligned} \epsilon(x) &= \{ @^\#(1, \mathbf{r}), @^\#(2, \mathbf{r}) \} & \eta(@^\#(1, \mathbf{r})) &= (\mathbf{AInt}, \emptyset, \emptyset) \\ \epsilon(\text{List } @^\#(2, \mathbf{r})) &= \{ @^\#(1, \mathbf{r}), @^\#(2, \mathbf{o}) \} & \eta(@^\#(2, \mathbf{r})) &= (\mathbf{AList}, \emptyset, \emptyset) \\ \epsilon(\text{List } @^\#(2, \mathbf{o})) &= \{ @^\#(1, \mathbf{r}), @^\#(2, \mathbf{o}) \} & \eta(@^\#(2, \mathbf{o})) &= (\mathbf{AList}, \emptyset, \emptyset) \end{aligned}$$

Containers & Polymorphism. The content variables of each container have a name defined by their abstract address, depending on the allocation site. This means that if a variable \mathbf{l} is assigned different lists in two different conditional branches (as in the example below), two different element variables will be created, and no polymorphic relationship will be inferred. To counter this issue, we start by unifying both abstract states before performing the join, renaming both element variables into a single one. Our analysis is then able to infer that x has the same type as the element of the list \mathbf{l} , which is either integers or strings:

```

1  if *: l = [1,2,3]
2  else: l = ['a', 'b', 'c']
3  x = l[0]

```

6 Implementation and Experimental Evaluation

6.1 Modular Implementation into Mopsa

We have implemented our analysis into Mopsa, a framework aiming at easing the development of static analyses by abstract interpretation [20, 24]. Mopsa currently supports the analysis of subsets of the C and Python programming languages. It is written in OCaml. The framework uses domain modules with a uniform signature to describe abstract domains, control-flow iterators, etc. This ensures that the domains are loosely coupled; they can be easily combined and reused. In addition, domains can rewrite expressions and statements dynamically, which makes it easier to reuse existing abstractions defined over a different syntax or semantics. For instance, Python loops are first rewritten into a canonical shape, while the fixpoint computation is handled by another more generic module, used to handle C loops as well. More details about Mopsa can be found in [20]. The type analysis consists in 2000 lines of OCaml code, the container abstraction consists in 1600 lines of OCaml, and there are 5000 lines of OCaml code defining the iterators and data model of Python.

6.2 Optimizations & Extensions

During our initial testing of our analysis, we noticed that it was slowed down by two factors: the number of exceptions that were raised (creating a large number of abstract states to store), and the analysis of function calls (where the same functions were analyzed many times). This led us to two optimizations described below. We then explain how we use Python type annotations to analyze more programs.

Exception Abstraction. When an exception is raised, we store the current abstract state with the exception flow token for the rest of the analysis, in order to reuse it if this exception is caught later on. However, unprecise analyses may raise exceptions frequently. For example, the smashing abstraction handling the list analysis needs (in order to be sound) to raise a potential `IndexError` at each list access, as the analysis does not keep track of the list size. This created many different exceptions stored in the analysis state, but most were never used. To solve the problem, we abstracted sets of such exceptions for which the analysis is deemed *a priori* unprecise (which can be parameterized by the user) into a single abstract exception, joining the corresponding abstract states into one. By default, the exceptions abstracted are `IndexError`, `KeyError` and `ValueError`.

Towards a Partially Modular Function Analysis. We have implemented a partially modular function analysis, which keeps the abstract input state, the abstract output state and the result of function calls in a cache. When analyzing a function call, the cache is checked: if this function has already been analyzed with the *same* abstract input state, the analysis result is taken directly from the cache. Otherwise, the function is inlined, and the analysis result cached afterwards. In particular, using this cache does not reduce the precision of the analysis, but greatly improves its running times. The experiments displayed in Table 16 show that this cache, combined with the exception abstraction can provide a 32x speedup over the inlining-based analysis (`regex_v8.py`), while the memory usage increased by 15%. In some cases, the inlining-based analysis and the cache-based analysis have the same running times: this may be due to a program having less user-defined functions to analyze, or the cache not being hit because the calling contexts are too different. We believe this cache is particularly efficient because we compute types rather than values: while the abstract state would change a lot during a value analysis (e.g, as loop indexes increase), the abstract state in the case of a type analysis is more stable.

We can also reuse the cache when the current input state is less than the input state kept in the cache. This is actually used in our implementation. In the benchmarks below, choosing this relaxed version improves the running times by 40% in one case (`choose.py`), but introduces imprecision in another case (22 out of the 25 alarms detected in `hexiom.py`).

Note that we keep analyzing functions on demand, at each call-site, knowing their calling context. We believe that performing a sound, context-free function call analysis, as done in most type systems, would not be practical for Python programs, as functions rely on implicit assumptions and may have side effects on their arguments or other variables not defined in the function scope. The cache-based analysis could still be improved to keep only the relevant parts of the whole abstract input and output states, such as the parts that may be read or changed by the function. This extension, which would help reuse more of the analysis results kept in the cache, is left as future work.

Using Type Annotations. As the Python standard library is huge, and partly written in C, we needed a way to support the C-written part without too much manual work. We decided to leverage the work from the `Typedsh` project [37], which offers type annotations for a substantial part of the standard library. This project uses the standard type annotations recently introduced by the PEP 484 into Python [36]. These type annotations are quite powerful (they feature possibly bounded polymorphism using `TypeVar`, structural subtyping support with `Protocol`, disjunctive function signatures with the `@overload` decorator, ...). For example, they can completely specify the signature of the `fspath` function described in the introduction:

```
T = TypeVar('T', str, bytes)
class PathL(Protocol[T]):
    def fspath(self) -> T: ...
@overload
def fspath(path: PathL[T]) -> T: ...
@overload
def fspath(path: str) -> str: ...
@overload
def fspath(path: bytes) -> bytes: ...
```

These annotations remain less expressive than our analysis, as side-effects (such as raised exceptions, aliasing) cannot be expressed yet, but a type-and-effect system [23] could be used. When a stubbed module is imported, our analyzer parses the corresponding annotated file and stores its functions (similarly for classes and variables). Then, when a stubbed function is called, we check that the arguments match the function signature. In that case, we assume that the function has no side effects and returns an object of the annotated return type, which we convert into an abstract object. Note that the use of these annotations changes the soundness of our analyzer: exceptions raised by concrete functions where we used their annotated counterpart will not be reported.

6.3 Experimental Evaluation

In this part, we evaluate our implementation on several benchmarks. We compare our analysis with four tools aiming at detecting incorrect programs potentially reaching runtime errors: the abstract-interpretation-based value analysis of Python [13], and three other tools having close goals: a tool by Fritz & Hage [12], `Typete` [16] and `Pytype` [42]. We also include the static analysis part of `RPython` [2] in our comparison, whose goal is to compile a restricted subset of Python into more efficient programs. [12, 16, 42] try to infer a static type that ensures the absence of dynamic typing errors, while we go further and check whether dynamic typing errors can occur and result in exceptions that stop the program (hence, we

can successfully analyze correct programs that are not typable but are nevertheless correct as they recover from dynamic type errors). Both [13] and our analyzer generate as output the set of exceptions that may escape to the toplevel, with detailed exception messages close to those given by Python. While this naturally includes type-related exceptions, we also take into account that even type errors can be caught and handled by the program, in which case they are not reported as errors. Contrary to [12, 16, 42], we also detect other errors (such as out of bound list accesses), in order to have a sound analysis (though we are unprecise in most cases). We also show the performance gain of the optimizations described in Section 6.2.

Competing Tools. The tool developed by Fritz and Hage performs a data-flow analysis, computing the type of each variable. While the original paper [12] experiments various tradeoffs between performance and precision (using different widenings, flow-sensitivity, context-sensitivity, . . .), we used the default arguments of the provided artifact. As mentioned in their paper, this tool does not handle exceptions nor generators. Its output is a dump of the data-flow map, associating to each program point the type of each variable. A program is untypable for this tool when the analyzer puts reachable variables to the bottom type.

Typpete encodes type inference of Python programs into a MaxSMT problem, and passes it to Z3 to solve it. If Z3 yields `unsat`, the program is untypable. Otherwise, the output of Typpete is a type annotation of the input program. It comes with around 40 examples on which we were able to test our analyzer. Typpete restricts its input to Python programs where variables have a single type in a program (but it handles subtyping: a variable having both types `int` and `str` will have type `object`) and dynamic attribute addition is not supported. When there is a type error, Z3 finds the inference problem to be unsatisfiable and Typpete shows a line in relationship with the type error. As the structure of the program is lost during the MaxSMT encoding, the line shown by Typpete is not always the line where the error will occur at runtime. Typpete supports the basics of the PEP 484 type annotations, and uses them for its stubs, or to guide the analysis on an input program.

Pytype is a tool developed by Google and actively used to maintain their codebase, hence it is more mature than the other tools. It performs an analysis that is not described formally, but it has a wide language and library support (it also uses Typeshed), allowing it to scale to large codebases. It outputs the last type of each variable when the typing is successful, and can produce a type annotation of the input program. It also produces clear error messages looking like the exceptions raised by Python when it detects an erroneous program.

We obtained the analyzer developed by Fromherz et al. [13]. It performs a value analysis by abstract interpretation. Its output is a set of potentially uncaught exceptions.


RPython performs a data-flow analysis to check that a program is part of the subset it can efficiently compile. It outputs the control-flow graph with the inferred types.




We compare the analysis of these five tools to two different configurations of our analyzer:

- **Conf. 1** using inlining and no exception smashing of the alarms;
- **Conf. 2** using partially modular analysis and exception smashing in alarms (see Sec. 6.2).

We have not noticed any improvement using the relational domain in the benchmarks, so the results below use the non-relational analysis. The relational analysis increases the analyses times by a factor 5 at most.

Benchmarks. We chose 5 of the biggest benchmarks from Typpete’s unit tests (prefixed with 🐍 in Table 16). We also took 12 benchmarks from Python’s reference interpreter [39] (prefixed with 🐍 in the table). Out of the 44 benchmarks currently available, we chose 12

with no external dependencies and few standard library module dependencies, so that most tools are able to analyze them. We argue that while the benchmarks are not very long, these Python programs are realistic and may call a lot of functions. For example, calling Python profiler `cProfile` on `chaos.py` shows more than 469,000 function calls. We also add three small tests focusing on characteristics we believe are paramount to performing a sound analysis of Python programs: taking into account object mutation and aliasing, as shown in Fig. 3 (file `mutation.py`); being able to precisely analyze introspection operators such as `isinstance` and `hasattr`, in order to analyze precisely a program calling the function `fspath` from Fig. 1 for example (file `isinstance.py`, Fig. 14); and analyzing precisely exception-related control-flow operators in order to have a precise analysis and avoid raising type errors later caught by an `except TypeError` statement for example (file `exception.py`, Fig. 15). Finally, we analyze the two main parts (`processInput.py`, `choose.py`) of a real-world command-line utility from Facebook, called PathPicker (prefixed with ; the LOC for these files consists in the size of the file and all the PathPicker files imported by this one). These two parts are multifile projects depending on other modules from PathPicker (which are inlined and analyzed by our tool), as well as some standard library modules, including `re`, `subprocess`, `json`, `curses`, `posixpath`, `argparse`, `configparser`, `os`, `stat`, `locale`, `bz2`, `lzma` respectively handling regular expressions, external process calls, json files, curses command-line interfaces, file-related functions, argument parsing, configuration file parsing, operating-system and file status, internationalization of output and compression algorithms. As all these modules are at least partially written in C, we used the annotations from Typeshed [37] to support them. All program constructs used in the benchmarks are supported by our tool, meaning our analysis is sound on them.


Performance and Precision Evaluation. We test the language support, the performance and the precision of each tool. An analyzer may crash due to an unsupported construction () , or may timeout after one hour of analysis (). We measured the analysis time five times for each benchmark and tool, and the mean is displayed. All tools are deterministic. In the evaluation of our tool in its most efficient configuration (Conf. 2), the column  displays the number of false alarms raised (the precision is identical in Conf. 1), with the smashed exceptions (corresponding to the unprecise exceptions raised by the list and dictionary abstractions) separated. The results are displayed in Table 16.

We notice that our analysis is able to scale to benchmarks a few thousand lines long, within a reasonable analysis time. Some benchmarks take longer to analyze: for example `hexiom.py` has a lot of nested loops, and functions are called multiple times, so the analyzer has a lot of fixpoint computations and inlining to perform (it performs 1770 analyses of 5-levels nested loops). It seems that the other type analyzers [12, 16, 42] do not perform fixpoint computations over loops (which at least for the case of Typypete seems sound as it infers more abstract types). Similarly, Typypete is able to perform an efficient analysis, although it lacks library support to analyze some Python benchmarks, and is unable to analyze programs where a variable is initialized in a (potentially unexecuted) loop. The

```

1 if isinstance(x, int): y = 4
2 else: y = 'a'
3 z = 2 + y


```

 **Figure 14** `isinstance.py`.

```

1 try: z = 2 + 'a'
2 except: z = 3.14
3 a = z+1

```

 **Figure 15** `exception.py`.

■ **Table 16** Analysis of Python benchmarks.

Name	LOC	Conf. 1	Conf. 2	▲	[12]	Pytype	Typpete	[13]	RPython
isinstance.py	3	42ms	40ms	0	1.2s	0.78s	0.67s	10ms	4.9s
exception.py	3	37ms	34ms	0	1.3s	0.70s	0.57s	9ms	✘
mutation.py	12	34ms	34ms	0	1.3s	0.75s	0.68s	11ms	✘
disjoint_sets.py	45	70ms	59ms	0 [†]	0.92s	0.91s	1.2s	✘	8.8s
functions.py	58	41ms	39ms	0 [†] [Ⓐ]	1.2s	0.84s	1.1s	✘	8.0s
fannkuch.py	59	76ms	69ms	0 [†]	1.2s	0.80s	✘	0.31s	✘
bellman_ford.py	61	0.17s	0.24s	0 [†]	1.4s	0.99s	1.4s	2.4m	7.1s
float.py	63	0.13s	82ms	0 [†]	1.7s	0.92s	1.3s	0.84s	5.6s
coop_concat.py	64	45ms	43ms	0 [†]	1.8s	0.81s	1.3s	20ms	✘
spectral_norm.py	74	0.32s	0.19s	1	1.6s	0.98s	✘	✘	✘
crafting.py	132	0.48s	0.41s	0 [†] [Ⓐ]	1.6s	0.97	1.7s	✘	✘
nbody.py	157	1.4s	0.80s	1 [†] [Ⓐ] [±]	1.7s	1.3s	✘	✘	✘
chaos.py	324	8.9s	2.3s	0 [†] [±]	13s	11s	✘	✘	✘
raytrace.py	411	3.5s	1.5s	7 [±]	36s	2.8s	✘	✘	✘
scimark.py	416	0.85s	0.55s	2 [†]	8.5s	4.4s	✘	✘	✘
richards.py	426	11s	5.0s	2 [†] [±]	38s	2.4s	✘	✘	7.8s
unpack_seq.py	458	13s	4.2s	0 [±]	1.1s	7.4s	2.7s	14s	✘
go.py	461	4.0m	15s	32 [†] [±]	8.5s	3.4s	✘	✘	✘
hexiom.py	674	6.9m	22s	25 [†] [Ⓐ] [±]	✘	4.2s	✘	✘	✘
regex_v8.py	1792	8.2m	15s	0 [†]	4.9s	⊙	1.7m	✘	✘
processInput.py	1417	6.1s	4.8s	7 [†] [Ⓐ] [±]	2.4s	11s	✘	✘	✘
choose.py	2562	8.6m	46s	17 [Ⓐ] [†] [±]	1.7s	15s	✘	✘	✘

✘ unsupported by the analyzer (crash) ⊙ timeout (after 1h)

Smashed Exceptions: KeyError [Ⓐ], IndexError [†], ValueError [±]

tool from Fritz and Hage is quite fast (the running times are measured by running a docker container due to the software dating from 2011), but we will see later that it is unsound in most cases. It fails on `hexiom.py` due to a parsing error. Pytype is a more mature analyzer, and it does not fail on any of the benchmarks, but times out in the `regex_v8.py` benchmark (after reaching out to Google, it appears to be a performance bug from Pytype in its analysis of big dictionaries). The value analysis [13] is unable to support the standard library functions needed for most benchmarks (supporting new library functions in the value analysis is more time-consuming, as it requires to include the effect of this function on the abstract values). On the benchmarks it is able to pass, our analysis is in average 8.5× faster than the value analysis; it also scales to benchmarks 5× longer. RPython is able to type 6 out of 22 benchmarks. In the 16 other cases, 5 seem to be due to internal bugs, while the 11 last cases are due to constructs unsupported by RPython. Compared to RPython, our analysis is able to fully analyze invalid programs (it will not stop at the first type exception, which can be caught later on).

Our analysis raises a few alarms (as all programs are correct, all alarms are false alarms here). As the programs did not mix types implicitly, our analysis was sufficiently precise to avoid raising false alarms over type and attribute errors. However, the smashing abstraction of the lists and the dictionaries creates some false alarms: dictionary values having different types (and heterogeneously-typed lists) are smashed into content variables, triggering imprecision over the types in the rest of the analysis. In addition, the smashing abstraction currently does

not keep track of the (potential) emptiness of lists: this creates a few alarms, as variables initially defined during a loop iteration over a list may be undefined in the rest of the program if the list is empty (raising `UnboundLocalError` in `spectral_norm.py`, `nbody.py`, `bm_raytrace.py`, `bm_scimark.py` and 22 in `hexiom.py`). More generally, the absence of information on the length of the list means that each list access should raise a potential `IndexError` (we could reduce the number of false alarms by adding a small domain keeping track of the abstract length of lists; this is left as future work). Similarly, `KeyError` are raised upon each dictionary access, and `ValueError` may be raised during list unpacking. The spurious `IndexError` are not raised by the value analysis [13], which is able to track the length of lists. The alarms are not raised by the three other analyzers, as they focus on type errors only, and not on finding which exceptions may be raised. As each analyzer has its own output, we were unable to compare their precision in all cases, and only study the precision on the first three small examples. In the `isinstance.py` example, both our tool, [13] and Pytype are precise, but the others are unprecise ([12] yields an unsound result, Typpete declares the program incorrect). For `exception.py`, [12] does not support exceptions; while [13], Pytype and Typpete declare the program incorrect; our tool does not raise any alarm. Concerning `mutation.py`, both Pytype, [13] and our tool are precise. Typpete is unprecise (it declares some integers and strings to be of object type), and [12] infers a variable holding a string as an integer.

Soundness Evaluation. We experimentally check the soundness of the analyzers. We believe soundness is important in order to detect all potential errors. As each benchmark file was a correct Python program, we created erroneous variants having one type error (by introducing a string into an integer variable), in order to check the soundness of each analyzer (similarly to the evaluation of Typpete). We then ran each analyzer on those files (the correct and the erroneous one each time), and checked whether the inferred types and alarms were matching the behavior of the program: either the analysis seemed sound as the types and alarms were correctly raised, or the analysis was unsound (no error was detected in the erroneous variant). The injection of type errors to evaluate the soundness is simplistic, as our goal was to quickly test the soundness of the other tools. Our analyzer is sound by construction but may include implementation bugs, and it would be interesting to automate error injection to experimentally check the soundness more thoroughly.

We find that our analysis catches the errors in all erroneous variants and is thus sound – as expected – in these cases. Typpete is sound over the programs it can analyze. The tool from Fromherz et al. is unsound in the case of `unpack_seq.py` due to an implementation error. The artifact from [12] is unable to detect errors in all cases except `fannkuch.py`. Pytype is not sound in a few cases (`bellman_ford.py`, `crafting_challenge.py`, `float.py`, `richards.py`, `spectral_norm.py`, `unpack_seq.py`).

Evaluation Summary. Our analysis is sound, it reports a few false alarms. Preliminary results indicate that our analyzer is able to scale, at least on programs a few thousand lines long. The soundness evaluation showed that even simple errors such as replacing an integer with a string may go unnoticed for unsound analyzers.

Comparatively, Pytype is the most advanced tool: it is able to scale and seems to support most of the standard library. It is however unsound in some cases. Both Typpete and [13] perform a sound analysis, but they lack some language or library support in the bigger benchmarks. The tool from Fritz and Hage is able to analyze programs very quickly, and supports most benchmarks, but it is unsound in most cases. RPython has a different goal, as it focuses on compiling a more static subset of Python efficiently. Most of the benchmarks use constructs too dynamic for RPython to compile them efficiently.

7 Related Work

In this section, we discuss related work, focusing on formal analyses for the two most popular dynamic programming languages: JavaScript and Python.

JavaScript. JavaScript is defined by a standard, and has been formalized in Coq [7] and in K [26]. [18] presents the first static analysis by abstract interpretation for JavaScript, and provides an implementation called TAJIS. [19] builds upon TAJIS to define a more efficient interprocedural analysis. As strings play a wide role in the semantics of JavaScript, precise string abstractions are studied in [1, 22]. [21] uses a static type analysis to optimize numerical computations datatypes. [17] proposes a method to soundly translate some `eval` statements into code, in order to improve the precision of their analysis. An analysis of asynchronous JavaScript built upon TAJIS is presented in [32].

Python. In [28], the authors define a mechanized semantics for a restricted subset of Python, consisting in basic values (integers, booleans) and control structures (loops, conditionals), but not taking objects into account. [31] proposes a semantics of Python 2.5 under the form of a Haskell interpreter. [27] defines a small-step semantics for a core Python language, λ_π , as well as a compiler from Python to λ_π , and a λ_π interpreter written in Racket. [15] shows a rewriting semantics for Python using the K framework [29]. [13] defines an interpreter-like semantics on which the concrete semantics presented in Section 2 is based.

Pyannotate [40] and MonkeyType [38] are tools performing a dynamic analysis: they collect the types of a Python program during its execution. Contrary to static analyses where an abstraction of the set of program traces is computed, dynamic analyses only run on one trace, meaning that non-determinism due to inputs or random choices will not be taken into account. While this approach helps developers move to type-annotated Python codes, the collected types correspond to one execution only, and are thus not sound.

A middle-end between dynamic and static type analysis is gradual typing [30, 14]. In that case, the programmer annotates parts of the program, which can then be typechecked. The unannotated parts of the program have an unknown type called `top`, from which any static type can be cast to and from. The soundness theorem of gradual typing then guarantees that if a program gradually typechecks, the only type errors that may occur at runtime are casts concerning variables having type `top`. Gradual typecheckers for Python include Mypy [35] and Pyre [41]. Both tools restrict the input language, as annotated variables can have only one type during the program execution (this type can be a union of types). By contrast, our type analysis is more permissive as it does not restrict the dynamic typing features of Python. We also do not require any annotation to run our analysis.

The closest approaches to our work [12, 42, 16, 2] have been described in the experimental evaluation (Sec. 6.3). It should be noted that Fritz & Hage [12] test many different parameter instantiations of their data-flow analysis. We believe that in the context of formal verification, a precise, context-sensitive, sound type analysis is useful. The flow-sensitivity is needed to precisely analyze exception catching statements, but neither have we tested this hypothesis on a larger scale nor have we tried selective flow-sensitivity, contrary to [12]. [34] presents a predictive analysis based on symbolic execution for Python. It consistently finds bugs and scales to projects of thousands of lines of codes, but it does not cover all executions, and is thus not sound. [13] performs a static value analysis by abstract interpretation. It uses abstract values similar to the ones presented in [18]. This analysis is not strictly more expressive than ours: while it focuses on values, it is relational over numerical datatypes,

but not over types. Our type analysis is more scalable in its implementation, as supporting new constructs consists in providing a type signature (and knowing the side effects of this function, including the potentially raised exceptions). To scale more quickly, we can also reuse `Typeshed` and its type annotations to support most of the standard library (though we will lose any side effect of the annotated function in that case). The type analysis also uses less memory and is quicker: we store type information rather than abstract values, and the fixpoint computations during the analysis of loops converge more quickly (types vary less than values, for example during loop iterations). The experiments of this value analysis consisted in some of Python's unit tests and some of Python's benchmarks. As the unit tests consist mostly in equality assertions over values, our type analyzer is unable to verify these. However, the running times for the type analysis on those tests are similar to the ones described in [13]. The benchmarks were shown in Table 16.

8 Conclusion

We have developed a static type analysis of Python programs by abstract interpretation, which collects uncaught exceptions that may be raised during a program execution. This analysis is sound, and its modular implementation scales to benchmarks a few thousand lines long. In addition, we found that compared to other type analyses, we uniquely take into account dynamic Python features such as object mutability, introspection operators, and exception-based control-flow statements.

Future work includes: speeding-up the inlining-based analysis with an efficient, summary-based function analysis; exploring the abstraction-precision trade-offs of the analysis (e.g. using an expansion-based container abstraction); analyzing bigger programs; combining this analysis with a value analysis (e.g. to keep track of the abstract length of summarized lists, in order to remove `IndexError`-based false alarms); finally, we will consider using the type information inferred by the analysis to optimize the execution of Python programs.

References

- 1 Roberto Amadini, Alexander Jordan, Graeme Gange, François Gauthier, Peter Schachte, Harald Søndergaard, Peter J. Stuckey, and Chenyi Zhang. Combining string abstract domains for JavaScript analysis: An evaluation. In *TACAS (1)*, volume 10205 of *LNCS*, pages 41–57, 2017.
- 2 Davide Ancona, Massimo Ancona, Antonio Cuni, and Nicholas D. Matsakis. Rpython: a step towards reconciling dynamically and statically typed OO languages. In *DLS*, pages 53–64. ACM, 2007.
- 3 Gogul Balakrishnan and Thomas W. Reps. Recency-abstraction for heap-allocated storage. In *SAS*, volume 4134 of *LNCS*, pages 221–239. Springer, 2006.
- 4 Kim Barrett, Bob Cassels, Paul Haahr, David A. Moon, Keith Playford, and P. Tucker Withington. A monotonic superclass linearization for dylan. In *OOPSLA*, pages 69–82. ACM, 1996.
- 5 Julien Bertrane, Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, and Xavier Rival. Static analysis and verification of aerospace software by abstract interpretation. *Foundations and Trends in Programming Languages*, 2(2-3):71–190, 2015.
- 6 Bruno Blanchet, Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. Design and implementation of a special-purpose static program analyzer for safety-critical real-time embedded software. In *The Essence of Computation*, volume 2566 of *LNCS*, pages 85–108. Springer, 2002.

- 7 Martin Bodin, Arthur Charguéraud, Daniele Filaretti, Philippa Gardner, Sergio Maffei, Daiva Naudziuniene, Alan Schmitt, and Gareth Smith. A trusted mechanised JavaScript specification. In *POPL*, pages 87–100. ACM, 2014.
- 8 Patrick Cousot. Types as abstract interpretations. In *POPL*, pages 316–331. ACM Press, 1997.
- 9 Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL*, pages 238–252. ACM, 1977.
- 10 Pascal Cuoq, Florent Kirchner, Nikolai Kosmatov, Virgile Prevosto, Julien Signoles, and Boris Yakobowski. Frama-C - A software analysis perspective. In *SEFM*, volume 7504 of *LNCS*, pages 233–247. Springer, 2012.
- 11 David Delmas, Eric Goubault, Sylvie Putot, Jean Souyris, Karim Tekkal, and Franck Védrine. Towards an industrial use of FLUCTUAT on safety-critical avionics software. In *FMICS*, volume 5825 of *LNCS*, pages 53–69. Springer, 2009.
- 12 Levin Fritz and Jurriaan Hage. Cost versus precision for approximate typing for Python. In *PEPM*, pages 89–98. ACM, 2017.
- 13 Aymeric Fromherz, Abdelraouf Ouadjaout, and Antoine Miné. Static value analysis of Python programs by abstract interpretation. In *NFM*, volume 10811 of *LNCS*, pages 185–202. Springer, 2018.
- 14 Ronald Garcia, Alison M. Clark, and Éric Tanter. Abstracting gradual typing. In *POPL*, pages 429–442. ACM, 2016.
- 15 Dwight Guth. A formal semantics of Python 3.3. Technical report, University of Illinois, 2013. URL: https://www.ideals.illinois.edu/bitstream/handle/2142/45275/Dwight_Guth.pdf?sequence=1&isAllowed=y.
- 16 Mostafa Hassan, Caterina Urban, Marco Eilers, and Peter Müller. MaxSMT-based type inference for Python 3. In *CAV (2)*, volume 10982 of *LNCS*, pages 12–19. Springer, 2018.
- 17 Simon Holm Jensen, Peter A. Jonsson, and Anders Møller. Remedying the eval that men do. In *ISSTA*, pages 34–44. ACM, 2012.
- 18 Simon Holm Jensen, Anders Møller, and Peter Thiemann. Type analysis for JavaScript. In *SAS*, volume 5673 of *LNCS*, pages 238–255. Springer, 2009.
- 19 Simon Holm Jensen, Anders Møller, and Peter Thiemann. Interprocedural analysis with lazy propagation. In *SAS*, volume 6337 of *LNCS*, pages 320–339. Springer, 2010.
- 20 M. Journault, A. Miné, R. Monat, and A. Ouadjaout. Combinations of reusable abstract domains for a multilingual static analyzer. In *Proc. of the 11th Working Conference on Verified Software: Theories, Tools, and Experiments (VSTTE19)*, pages 1–17, July 2019.
- 21 Francesco Logozzo and Herman Venter. RATA: Rapid atomic type analysis by abstract interpretation - application to JavaScript optimization. In *CC*, volume 6011 of *LNCS*, pages 66–83. Springer, 2010.
- 22 Magnus Madsen and Esben Andreasen. String analysis for dynamic field access. In *CC*, volume 8409 of *LNCS*, pages 197–217. Springer, 2014.
- 23 Daniel Marino and Todd D. Millstein. A generic type-and-effect system. In *TLDI*, pages 39–50. ACM, 2009.
- 24 A. Miné, A. Ouadjaout, and M. Journault. Design of a modular platform for static analysis. In *Proc. of 9th Workshop on Tools for Automatic Program Analysis (TAPAS'18)*, LNCS, page 4, 28 August 2018.
- 25 Hakjoo Oh, Kihong Heo, Wonchan Lee, Woosuk Lee, and Kwangkeun Yi. Design and implementation of sparse global analyses for C-like languages. In *PLDI*, pages 229–238. ACM, 2012.
- 26 Daejun Park, Andrei Stefanescu, and Grigore Rosu. KJS: A complete formal semantics of JavaScript. In *PLDI*, pages 346–356. ACM, 2015.

- 27 Joe Gibbs Politz, Alejandro Martinez, Matthew Milano, Sumner Warren, Daniel Patterson, Junsong Li, Anand Chitipothu, and Shriram Krishnamurthi. Python: The full monty. In *OOPSLA*, pages 217–232. ACM, 2013.
- 28 Ranson, Hamilton, and Fong. A semantics of Python in Isabelle/HOL. Technical report, University of Regina, 2008. URL: <http://www.cs.uregina.ca/Research/Techreports/2008-04.pdf>.
- 29 Grigore Rosu and Traian-Florin Serbanuta. An overview of the K semantic framework. *J. Log. Algebr. Program.*, 79(6):397–434, 2010.
- 30 Jeremy G. Siek and Walid Taha. Gradual typing for objects. In *ECOOP*, volume 4609 of *LNCS*, pages 2–27. Springer, 2007.
- 31 Gideon Joachim Smeding. An executable operational semantics for Python. *Universiteit Utrecht*, 2009. URL: <http://gideon.smdng.nl/wp-content/uploads/thesis.pdf>.
- 32 Thodoris Sotiropoulos and Benjamin Livshits. Static analysis for asynchronous JavaScript programs. In *ECOOP*, volume 134 of *LIPICs*, pages 8:1–8:30. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2019.
- 33 Fausto Spoto. The julia static analyzer for Java. In *SAS*, volume 9837 of *LNCS*, pages 39–57. Springer, 2016.
- 34 Zhaogui Xu, Peng Liu, Xiangyu Zhang, and Baowen Xu. Python predictive analysis for bug detection. In *SIGSOFT FSE*, pages 121–132. ACM, 2016.
- 35 Mypy. <http://mypy-lang.org/>, 2018. Accessed: 2018-07-22.
- 36 Python enhancement proposal 484, about type hints. <https://www.python.org/dev/peps/pep-0484/>, 2018. Accessed: 2018-07-23.
- 37 Typeshed. <https://github.com/python/typeshed/>, 2018. Accessed: 2018-07-22.
- 38 Monkeytype. <https://github.com/Instagram/MonkeyType>, 2019. Accessed: 2019-10-22.
- 39 Performance benchmarks from Python’s reference interpreter. <https://github.com/python/pyperformance/>, 2019. Accessed: 2019-10-22.
- 40 Pyannotate. <https://github.com/dropbox/pyannotate>, 2019. Accessed: 2019-10-22.
- 41 Pyre-check. <https://github.com/facebook/pyre-check>, 2019. Accessed: 2019-10-22.
- 42 Pytype. <https://github.com/google/pytype>, 2019. Accessed: 2019-10-22.
- 43 Pathpicker. <https://github.com/facebook/pathpicker/>, 2020. Accessed: 2020-01-03.

Reference Mutability for DOT

Vlastimil Dort 

Charles University, Prague, Czech Republic
dort@d3s.mff.cuni.cz

Ondřej Lhoták 

University of Waterloo, Canada
olhotak@uwaterloo.ca

Abstract

Reference mutability is a type-based technique for controlling mutation that has been thoroughly studied in Java. We explore how reference mutability interacts with the features of Scala by adding it to the Dependent Object Types (DOT) calculus. Our extension shows how reference mutability can be encoded using existing Scala features such as path-dependent, intersection, and union types. We prove type soundness and the immutability guarantee provided by our calculus.

2012 ACM Subject Classification Software and its engineering → Formal language definitions; Software and its engineering → Object oriented languages

Keywords and phrases Reference Mutability, Read-only References, DOT Calculus

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2020.18

Funding This work was partially supported by the Czech Science Foundation project 18-17403S, partially supported by the Mobility Fund of Charles University, and partially supported by the Charles University institutional funding project SVV 260588. This research was supported by the Natural Sciences and Engineering Research Council of Canada.

1 Introduction

The Scala programming language integrates functional and object-oriented programming, making available many of the benefits of both paradigms. One important benefit of purely functional programming is referential transparency, which makes reasoning about program behaviour easier, both for human programmers and for automated optimizers and verifiers. Nevertheless, Scala does not provide any verifiable way to specify which parts of a program are purely functional. Purity is an absence of all side effects; in this paper, we focus on mutation of objects in the heap as one specific but important side effect.

Reference mutability, also called reference immutability [19, 10], has been studied especially in Java as a way to control mutation. References to objects are classified as either read-write or read-only, and writes to fields through a read-only reference are forbidden. This applies transitively: when a reference is read from the field of an object through a read-only reference, the newly-read reference is made read-only as well. As a result, if all parameters of a function are read-only (and if there are no accesses to global variables), the function must be pure in the sense that it cannot modify any state in the heap that existed before it was called, although it does have the ability to allocate and mutate new objects.

As in related work [19, 20], we distinguish reference mutability from the stronger guarantee of object immutability: a reference mutability system like ours ensures that an object is not mutated through a read-only reference, but it is still possible for the object to change if it is aliased by other, read-write references. For the same reason, we avoid calling references *mutable* or *immutable*, since it is not the reference that can be mutated, but the object that it refers to, and even the referent of a read-only reference is not necessarily immutable. By the same reasoning, it would be more accurate to speak of the *read-only-ness* of a reference rather than of its mutability, but we use the latter term because the former is awkward.



© Vlastimil Dort and Ondřej Lhoták;

licensed under Creative Commons License CC-BY

34th European Conference on Object-Oriented Programming (ECOOP 2020).

Editors: Robert Hirschfeld and Tobias Pape; Article No. 18; pp. 18:1–18:28

Leibniz International Proceedings in Informatics



LIPICs Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Our goal is to bring read-only references to Scala. An empirical study [8] showed that about 35 to 70 percent of classes in large Scala codebases are either deeply or shallowly immutable. One challenge is the complexity of Scala and its type system relative to Java, and the interaction of reference mutability with Scala language features. In particular, Scala programs frequently use nested functions that have access to variables in outer scopes. A second challenge is that for maintainability and ease of adoption, we seek a system that integrates well with Scala’s existing type system. Reference mutability implementations for Java add entirely new type systems on top of Java’s type system. Since Scala’s type system already provides powerful and expressive features, it ought to be possible to use those features to implement at least parts of a reference mutability system, and we explore the feasibility of such an approach. By reusing existing features as much as possible, we aim for an implementation that would require few changes to an existing Scala compiler so that it could be easily maintained as the compiler evolves.

When we began this project, we explored designs by prototyping them in the Dotty compiler for Scala 3. The subtle conceptual errors that we encountered revealed the need for a more principled approach. Our contribution in this paper is a formalization of our design as an extension of the Dependent Object Types (DOT) calculus [2, 16, 14], a core calculus modelling the essence of Scala. We prove type soundness and the immutability property guaranteed by our extended DOT calculus, which we call roDOT. roDOT can serve as a foundation for a correct implementation of reference mutability for the full Scala language.

The rest of the paper is organized as follows. In Section 2, we present a baseline DOT calculus that we will extend with reference mutability. We overview our approach in Section 3: we identify the requirements needed from a type system to implement a useful reference mutability system, discuss how the features of DOT can partially satisfy the requirements, and introduce the changes that we make to DOT to fulfill the requirements. In Section 4, we present roDOT, the formal DOT calculus extended with reference mutability. In Section 5, we define properties of roDOT, namely type soundness and the immutability guarantee, and discuss their proofs. We survey related work in Section 6 and conclude in Section 7.

2 Baseline DOT

In this section, we present a baseline DOT calculus that we will later extend with reference mutability. There are multiple variants of DOT calculi in the literature. Our baseline is close to kDOT [12], which in turn is based Wadlerfest DOT [2], with the following main differences. In Wadlerfest DOT, objects are immutable. The semantics operates on plain program terms, with objects in the heap represented by let-binding terms. In kDOT, objects have mutable fields, which can be re-assigned to hold a reference to another object on the heap. The semantics of kDOT operates on configurations with an explicit heap, a program term being reduced, and a stack of continuations. We adopt these features of kDOT in our baseline DOT. In addition, kDOT defines constructors to model the gradual initialization of the fields of each object, like in Scala. We omit constructors from our baseline DOT because they are not necessary to model reference mutability. Thus, our baseline DOT is a variant of kDOT without constructors.

The syntax of the baseline DOT is in Figure 1. As in WadlerFest DOT and kDOT, the literals are objects and lambda abstractions. An object has a self parameter s (modelling the `this` keyword in Scala) and a sequence d of member definitions. Object members are either fields, which store a term that is reduced after each read of the field, or type members. As in kDOT, a literal can only appear in a let-binding of the form `let $z = l$ in t` , which ensures that every literal can be referred to by some variable z .

$x ::=$	Variable	$l ::=$	Literal	$T ::=$	Type
z	local	$\nu(s : T)d$	object	\top	top
s	self	$\lambda(z : T)t$	lambda	\perp	bottom
y	location	$t ::=$	Term	$\forall(z : T_1)T_2$	function
$d ::=$	Definition	νx	var	$\mu(s : T)$	recursive
$\{a = t\}$	field	$\text{let } z = t_1 \text{ in } t_2$	let	$\{a : T_1..T_2\}$	field decl
$\{A = T\}$	type	$\text{let } z = l \text{ in } t$	let-lit	$\{A : T_1..T_2\}$	type decl
$d_1 \wedge d_2$	aggregate	$x_1.a := x_2$	write	$x.A$	projection
$\Gamma ::=$	Context	$x.a$	read	$T_1 \wedge T_2$	intersection
	empty	$x_1 x_2$	apply		
$\Gamma, x : T$	binding				

■ **Figure 1** Baseline DOT syntax.

$\Sigma ::=$	Heap	$c ::=$	Configuration
\cdot	empty heap	$\langle t; \sigma; \Sigma \rangle$	
$\Sigma, y \rightarrow l$	heap object	$Q ::=$	Member type
$\sigma ::=$	Stack	$\{a : T..T\}$	tight field decl
\cdot	empty stack	$\{A : T..T\}$	tight type decl
$\text{let } z = \square \text{ in } t :: \sigma$	let frame	$R ::=$	Record type
$F ::=$	Inert context	Q	member
	empty	$R_1 \wedge R_2$	intersection
$F, y : S$	binding	$S ::=$	Inert type
		$\forall(z : T_1)T_2$	function
		$\mu(s : R)$	object

■ **Figure 2** Baseline DOT runtime syntax.

In the baseline DOT, we distinguish between local variables z that are bound by let terms and lambdas, self variables s that are bound by object literals, and heap locations y that represent addresses in the runtime heap. Heap locations cannot appear in the surface syntax; they are created only in runtime configurations in the operational semantics.

A program is expressed as a term, which, if correctly typed, reduces to a location of a single item on the heap. The meanings of the terms are standard. We use the notation νx to make it explicit when a variable is used as a term. A let term evaluates one term and substitutes the result into another term. Terms in DOT are in A-normal form (ANF), in that subterms of a term are generally variables, not arbitrary terms. Thus, every non-trivial term must be evaluated and assigned to a variable in a let binding before it can be used in a later term. A write term changes the value of a field of an object on the heap. A read term returns the last value written to a field, or the term value given to the field when the object was created. An apply term applies a lambda by substituting the argument into its body.

Types must be explicitly specified in lambda abstractions and object literals. Lambdas have a function type, which allows them to be applied in an apply term. Objects have a recursive type containing an intersection of field or type declaration types corresponding to the definitions forming the object. The recursive type allows the declarations to refer to other members of the object. An intersection type is a common subtype of two types. As in kDOT, a field declaration type specifies two types for a field, a setter and a getter type. The

getter type is given to a read term that reads the field, while the setter is the type a variable must have so that it can be written to the field. A type declaration type specifies the lower and upper bounds for a type member, which can be referred to by type selection. The top type is a supertype of every type; the bottom type is a subtype of every type.

Evaluation of a program is described as reduction of an initial configuration, which consists of a term, an empty stack, and empty heap. The heap binds locations y to literals l . The heap is modified by creating a new object using the let-lit term, or by changing the value of a field using the write term. In a final configuration, the stack is empty and the term is in normal form – a location of a single item on the heap.

In a typed configuration, *heap correspondence* ensures that for each location in the context, the heap contains a function or an object of the specified type. For objects it means that if $\Gamma(y) = \mu(s : R)$, then $\Sigma(y) = \nu(s : R)d$, where $\Gamma \vdash d : [y/s]R$. The type R is syntactically the same in Γ and Σ .

Type soundness ensures that evaluation of a typed term either progresses indefinitely or reaches a final configuration. A key ingredient of the type soundness proof is the definition of inert typing contexts. A concrete object in the heap holds a specific term in each field and a specific type in each type member, so it is possible to type such an object with a type in which all member types are tight: each type declaration type $\{A : T..T\}$ has equal upper and lower bounds T and each field declaration type $\{a : T..T\}$ has equal getter and setter types T . Such types with tight bounds are called inert, and many theorems about DOT calculi hold only in typing contexts containing only inert types [14]. A progress theorem proves that if a configuration can be typed in a typing context that gives an inert type for each object in the heap, then it is in a normal form or steps to another configuration. A preservation theorem proves that the resulting configuration after the step can still be given the same type in an inert context that corresponds to the possibly updated heap.

3 Overview

In this section, we examine how the baseline DOT system can be extended to support read-only references and overview the path to roDOT.

3.1 Requirements

A type system for reference mutability should satisfy the following requirements:

Keeping expressiveness. The type system should admit programs that do not use the new features similarly to the baseline DOT.

Mutability constraints. Additionally, the type system should provide a way to distinguish between read-write and read-only references. Read-write references should be convertible to read-only references, but not the other way. In a reference mutability type system, the distinction can be achieved by making each type read-write or read-only, with read-write a subtype of read-only.

Integration with type system features. The extensions should use existing features of the DOT type system where possible, and not interfere with them.

Type soundness. The extensions should not make the type system unsound - as in kDOT, each typed program should reduce to an answer or run indefinitely.

Guarantee of immutability. The type system should guarantee that only read-write references are used to mutate objects. This guarantee should be transitive: starting from a read-only reference, the system should prevent mutation of any other objects reached by any sequence of field reads. To achieve this, if a read term reads from a field of an object

through a read-only reference, the result of the read should be given a read-only type, even if the field contains a read-write reference. This change in the type of the reference is called viewpoint adaptation [10].

Mutability polymorphism. Previous work [19, 10] demonstrated the importance of methods that are polymorphic in the mutability of the receiver. Consider a getter method that reads a field of an object. When called on a read-only reference, the method can obtain only a read-only reference from the field due to viewpoint adaptation, and thus its return type must be read-only. But, when the same method is called on a read-write receiver, it can read a read-write reference from the field, and its return type should reflect this.

3.2 Example

As an example, we will encode an object with a field a and getter and setter methods m_g and m_s for that field. In Scala, such an object would be created by instantiating a class:

```
class C {
  var a: T = _
  def m_s(z: T): Unit = {a = z}
  def m_g: T = a
}
```

In the baseline DOT, such an object can be created by a let statement:

let $z = \nu(s : T_o)\{a = x\} \wedge \{m_s = \text{let } z_1 = \lambda(z : T)s.a := z \text{ in } z_1\} \wedge \{m_g = s.a\}$ in t , where $T_o \triangleq \mu(s : \{a : T\} \wedge \{m_s : \forall(z : T)\top\} \wedge \{m_g : T\})$ and x is an initial value of type T .

The m_s method mutates the contents, so a reference mutability type system should prevent calling it on a read-only reference. The m_g method should be polymorphic in the mutability of the receiver. After we present the roDOT calculus, we will show how this example can be encoded in it in Section 4.5.

3.3 Changes to the Calculus

We now summarize the changes to the baseline DOT to support read-only references. We motivate each change briefly here, but defer a detailed justification to Section 4.

3.3.1 Mutability Types

First, we must decide how to distinguish read-write and read-only types. A straightforward way is to define some special marker type M to designate read-write references. Any type can be made read-write by intersecting it with M . This satisfies our requirement that a read-write type should be a subtype of the read-only version of the same type.

We can test whether a type is read-write by testing whether it is a subtype of M . For a reference x , we define an operation $\Gamma \vdash \text{isrw } x$ as a typing judgment $\Gamma \vdash x : M$. To make a read-write version of an existing type T , we define an operation $\text{rw}(T) \equiv T \wedge M$.

3.3.2 Dependent Mutability

Second, we need some mechanism to implement mutability polymorphism. We will use a dependent mutability type, defined to have the same mutability as some specified reference.

We can achieve this with a careful choice of the read-write marker type: we reserve a type member name M for mutability, and choose $M \equiv \{M : \perp \dots \perp\}$. This choice makes it possible for a type to depend on the mutability of a reference x using the type selection $x.M$. The type $\{M : \perp \dots x.M\}$ is read-write (in the sense of being a subtype of $\{M : \perp \dots \perp\}$) if (and in an inert context only if) the reference x is read-write (has type $\{M : \perp \dots \perp\}$).

3.3.3 Viewpoint Adaptation

Third, the type system requires an operation $\Gamma \vdash x \triangleright T \rightarrow T'$ that performs the viewpoint adaptation described above. Given a reference x and a type T of a field, it returns T if x is a read-write reference, but a read-only version of T if x is a read-only reference. This operation could also be composed from two simpler operations: making a read-only version of T and combining the mutability of T and x .

While a read-write version of a given type can be made with a simple intersection, the opposite of making a read-only version cannot be done using any of the type operations in the baseline DOT. If it is already the case that $T <: \{\mathbf{M} : \perp.. \perp\}$, none of the operations removes this subtyping relationship while otherwise keeping T unchanged. Therefore we need a new relation **ro** that makes a read-only version of a type. We will define $\Gamma \vdash T \mathbf{ro} T'$ by recursion on the syntax of T , so that T' is a supertype of T , but not a subtype of $\{\mathbf{M} : \perp.. \perp\}$.

We also need to combine the mutabilities of x and T . The mutability of x can be expressed using the type selection $x.M$, but to determine the mutability of T , we need to define another relation **mu** similar to **ro**. In $\Gamma \vdash T \mathbf{mu} T'$, the type T' is a lower bound for the special type member \mathbf{M} given by T . We will define the **ro** and **mu** relations in detail in Section 4.2.

Using these new type operators, we can define viewpoint adaptation as $\Gamma \vdash x \triangleright T \rightarrow T_r \wedge \{\mathbf{M} : \perp.. T_m \vee x.M\}$, where $\Gamma \vdash T \mathbf{ro} T_r$ and $\Gamma \vdash T \mathbf{mu} T_m$. Notice that the upper bound $T_m \vee x.M$ is a union type. To implement viewpoint adaptation, we therefore need to extend DOT with union types. Union types are a feature of Scala 3 and were studied in some variants of DOT [16, 3], but not in kDOT, from which our baseline DOT is derived.

Note that even with union types added, **ro** cannot be implemented by a simple union $T \vee \mathit{not}M$ of T with some fixed read-only type $\mathit{not}M$, because the set of members of such a union type is the intersection of the members of T and $\mathit{not}M$, so the union type would not have all the members of T .

3.3.4 Recursive Types

If we were to allow the self type T in a recursive type $\mu(s : T)$ to be read-write, an object of such a type would be inherently mutable, i.e., viewpoint adaptation would not be able to create a read-only reference to it. This is because the read-write type $\mu(s : T \wedge \{\mathbf{M} : \perp.. \perp\})$ is not a subtype of the read-only variant $\mu(s : T)$, for there is no subtyping between recursive types in DOT.

This means that the mutability of an object must be expressed *outside* the recursive type as $\mu(s : T) \wedge \{\mathbf{M} : \perp.. \perp\}$, as opposed to inside as $\mu(s : T \wedge \{\mathbf{M} : \perp.. \perp\})$.

We also require the self type T to not refer to $s.M$, the mutability of the self variable. Otherwise, the mutability could be stored in a type member such as $\{A : s.M.. \perp\}$, from which we could infer $s.M <: \perp$, which would again make the object inherently mutable.

3.3.5 Methods

In a type of a mutability-polymorphic method, such as the getter m_g from the example, we want to specify its return type to be dependent on the mutability of the receiver. When the method is called on a read-write receiver reference, the type of the return value will become read-write as well, because of the mutability-dependent return type.

In the baseline DOT, a method is encoded as a function value stored in a field of an object. Given that the declaration of a field is typed with a self-variable s in scope, it would seem natural to use $s.M$ for defining methods with return types polymorphic in the mutability of

the receiver. For example, $\{m : \forall(z : \top)(\{a : \top\} \wedge \{M : \perp..s.M\})\}$ would be a method that returns a read-write reference to an object with field a when called on a read-write receiver, but returns a read-only reference to the same object when called on a read-only receiver.

The problem with this encoding is that the dependent mutability $s.M$ in the return type would refer to the mutability of the *object* that the method is contained in, not to the mutability of the *reference* to that object through which the method is called. Distinguishing these two concepts requires a rather complicated example, which we will present in Section 4.1.

To distinguish these concepts in roDOT, we introduce a new kind of variable r to represent the receiver reference and write it as an explicit additional parameter of each method. The type given to this parameter decides its mutability. In polymorphic methods, we type it as read-only, so that the method can be called on either a read-write or a read-only receiver.

Furthermore, the baseline DOT splits the typing of a method invocation into two steps: the first step reads the function value from the field and the second step applies the function to an argument. This two-step process separates the receiver reference, which is present only in the first step, from the function application in the second step. Thus, the mutability of the result can no longer polymorphically depend on the mutability of the receiver reference.

To overcome this problem, we need to unify method selection and method invocation into one step, so that the type of the method invocation can depend on the type of the receiver. We extend the baseline DOT with an explicit method construct. A method is called in a single step using a term of the form $x_1.m x_2$, which selects the method from the receiver x_1 and applies it to an argument x_2 . A method type then has the form $\{m(z : T_1, r : T_3) : T_2\}$.

Visibility

If a method captures a variable from its surrounding environment, it can write to the object that the variable refers to even if it is called on a read-only receiver and with a read-only argument. To prevent this, we hide variables other than the receiver and method parameter in the typing context when typing the body of a method, so the method cannot capture them. Despite this restriction, it is still possible to encode a method that captures variables as follows: the captured variables are copied into fields of the containing object. This workaround ensures that viewpoint adaptation is applied when the captured variable is read out of the field of the object.

3.3.6 Reference Variables

In the baseline DOT, a reference value is a location y , the unique identifier of some item in the heap. To state and prove roDOT immutability guarantee, we need to distinguish read-only and read-write references to the same object in a runtime configuration. We therefore extend the calculus with references w . Two references w, w' may designate the same location y , but can have different mutabilities in the typing context Γ . Runtime configurations are extended with an environment ρ that maps each reference w to the location y that it designates.

In summary, the baseline DOT distinguishes three kinds of variables: local variables and method parameters z , recursive self variables s , and heap locations y . To these three, roDOT adds receiver variables r and reference variables w .

4 Type System

In this section, we present the formal definition of roDOT.

4.1 Syntax

The syntax of roDOT is defined in Figure 3. Changes from the baseline DOT are highlighted using shading.

Terms are formed by the same syntax as in the baseline DOT, except that the function application syntax is replaced by a method call syntax and lambda literals are replaced by method definitions with the ordinary parameter z and receiver parameter r . Since the calculus no longer needs lambda literals, all literals are objects, so we inline them into the let-lit term. Furthermore, the values of fields are variables x rather than arbitrary terms t . Terms were needed in the baseline DOT to allow fields to hold lambdas to encode methods.

Variables x are classified as either abstract variables u , which are bound in terms and definitions in the initial program, or global variables v , which are generated during reduction and are used in the heap and the runtime environment. They are not expected to be used in the initial term. Abstract variables are either general local variables z , object self variables s , or method call receivers r . Global variables are either heap locations y or references to heap locations w . A typing context Γ can give a type to variables of any kind.

To support viewpoint adaptation, we add union types. They are dual to intersection types and make it possible to define a distributive subtyping rule for intersections of type member types (ST-TypAnd in Figure 4), which makes it possible to combine multiple mutability declarations into one. A new type \mathbf{N} is a read-only version of \perp .

$x ::=$	Variable	$\Gamma ::=$	Context
u	abstract		empty
v	global	$\Gamma, x : T$	binding
$u ::=$	Abstract	$\Gamma, !$	hide
z	local	$d ::=$	Definition
s	self	$\{a = x\}$	field
r	receiver	$\{m(z, r) = t\}$	method
$v ::=$	Global	$\{A(r) = T\}$	type
y	location	$d_1 \wedge d_2$	aggregate
w	reference	$T ::=$	Type
$t ::=$	Term	\top	top
νx	var	\perp	bottom
$\text{let } z = t_1 \text{ in } t_2$	let	$\mu(s : T)$	recursive
$\text{let } z = \nu(s : T)d \text{ in } t$	let-lit	$\{a : T_1..T_2\}$	field decl
$x_1.a := x_2$	write	$\{m(z : T_1, r : T_3) : T_2\}$	method decl
$x.a$	read	$\{B(r) : T_1..T_2\}$	type decl
$x_1.m x_2$	call	$x_1.B(x_2)$	projection
$B ::=$	Type name	$T_1 \wedge T_2$	intersection
A	type member	$T_1 \vee T_2$	union
\mathbf{M}	mutability	\mathbf{N}	read-only \perp

■ **Figure 3** Syntax.

4.1.1 Methods

Method declarations bind the parameter variable z and the receiver variable r . In the corresponding definition, we omit the types, because they are not needed. Each method has only one parameter other than the receiver. Multiple values can be passed to a method by wrapping them in an object and passing a reference to the object as the argument.

To motivate the dedicated syntax for methods, consider an object that contains a method a that returns a reference with the same mutability $s.M$ as the receiver that it is called on. In the syntax of the baseline DOT, this object could have the type $T \triangleq \mu(s : \{a : \forall(z : T')\{M : \perp..s.M\}\})$. Suppose x_1 is a read-write reference to such an object; it has type $T \wedge \{M : \perp.. \perp\}$. Now suppose x_1 is copied to another reference x_2 that is read-only. The reference can be made read-only in various ways: one way is to store x_1 into a field of some other object y , and then read it back into x_2 through a read-only reference to y , so viewpoint adaptation will make the type of x_2 read-only. But, even though x_2 is read-only (i.e., $x_2.M$ is not a subtype of \perp), x_2 still has the same field types copied from x_1 . In particular, x_1 has the type $T \wedge \{M : \perp.. \perp\}$, the type T , the type $\{a : \forall(z : T')\{M : \perp..x_1.M\}\}$ (by VT-RecE), and the type $\{a : \forall(z : T')\{M : \perp.. \perp\}\}$ (by ST-SelU). Even though x_2 is read-only, it still also has the latter field types. Thus, the expression $x_2.a$ can be typed as a function with a read-write return type, even though the receiver x_2 is read-only.

If we introduced methods, but without the receiver variable r , the type of the object would be $T \triangleq \mu(s : \{m(z : T') : \{M : \perp..s.M\}\})$. Suppose again that x_1 is a read-write reference to the object, which is copied to x_2 and made read-only. As before, the VT-RecE rule can be applied to the type of x_1 before it is made read-only, so x_1 has the type $T \wedge \{M : \perp.. \perp\}$, the type T , the type $\{m(z : T') : \{M : \perp..x_1.M\}\}$, and the type $\{m(z : T') : \{M : \perp.. \perp\}\}$. Even though x_2 is a read-only reference, it still also has the latter method types, and thus the method can return a read-write reference even when called on the read-only receiver x_2 .

To avoid these problems, we prohibit references to the mutability $s.M$ of the self object reference s in all definitions; the return type can only refer to the mutability of r , the receiver reference on which the method will be called.

One may wonder whether we could have achieved the same thing without changing the baseline DOT syntax by encoding a method with receiver r and parameter z using currying as $\{a = \lambda(r : T_3)\lambda(z : T_1)t\}$. The method would then be called on receiver r with argument x as $(r.a r) x$, i.e., the receiver r would have to be repeated. The problem with this encoding is that in the type of an object, we have to write the types of the object's methods, and those method types contain the type for the receiver, which should be the type of the object itself. Thus, the type of an object would have to recursively include itself, and thus the structure of the type would be infinite. On the other hand, with the explicit syntax for method types ($\{m(z : T_1, r : T_3) : T_2\}$), we can resolve this issue in the typing rule for method definitions: when we add the receiver r to the typing context for typing the method body, we add it not just with the specified type T_3 , but with an intersection of T_3 with the self type of the object containing the method. This removes the need to recursively repeat that self type inside T_3 .

4.1.2 Type Members

Type members can have either an ordinary name A , or the special mutability member name M , which cannot be defined in an object literal. We write B in places where both A and M can be used. In the formal syntax, all type members have a receiver parameter r with no type specified: the general form is $\{B(r) : T_1..T_2\}$. In a type selection, an argument for this parameter must be provided, and is substituted into the bounds T_1 and T_2 . To reduce clutter, we omit writing the receiver parameter when it is not used in the bounds.

18:10 Reference Mutability for DOT

The M is a special member used as the mutability marker. A type $\{M : \perp..T\}$ is read-write if $T <: \perp$ and read-only otherwise. Only the upper bound of M is significant for mutability because the lower bound is always \perp . A dual encoding would be equally possible, in which the lower bound would be significant and the upper bound would always be \top , so a read-write type would be expressed by $\{M : \top..T\}$.

The receiver parameter r allows making a generic method type, where the mutability of the result is determined by a type member of the containing object. For example, the mutability of the return type of the method in $\mu(s : \{m(z : \top, r : \top) : s.A(r)\})$ depends on A . It is read-write if the object defines $\{A(r) = \{M : \perp..\perp\}\}$, read-only if the object defines $\{A(r) = \{M : \perp..\top\}\}$, and polymorphic if the object defines $\{A(r) = \{M : \perp..r.M\}\}$.

4.2 Typing

The typing and subtyping rules are shown in Figures 4 to 6. The rules are obtained from the baseline DOT by applying the syntactic changes and by adding additional rules. As in the baseline DOT, typing rules for variables are separated from typing rules for terms.

The typing rules apply both to determine which initial terms are valid programs and to give types to intermediate terms during reduction in order to prove type safety. When used for this second purpose, the terms may contain type selections on reference variables such as $w.A$. For two references w and w' to the same location y , types $w.A$, $w'.A$ and $y.A$ are considered equivalent. In order to achieve this, subtyping and typing statements have the environment ρ on the left-hand side. The environment is passed around in all the typing rules, but only used in ST-Eq, which states that if two types only differ in references, then they are subtypes. It has no effect on typing of initial program terms, because those do not contain references w and are typed with empty ρ .

4.2.1 Subtyping Rules

The ST-Refl and ST-Trans rules ensure that subtyping is a preorder, and the ST-Top and ST-Bot rules establish \top and \perp as the maximum and minimum elements. The ST-Or* and ST-And* rules define the usual properties of unions and intersections. ST-Dist makes them distribute and ST-TypAnd allows merging bounds of type members in intersections.

The ST-Typ, ST-Met and ST-Fld rules allow subtyping between declaration types. In ST-Met, the parameter is in the typing context for subtyping of the receiver types, and both the parameter and receiver are in the context for subtyping of the result types. In ST-Typ, the r parameters do not have bounds and are not added to the typing context for subtyping. It is important for the proofs that in the tight-subtyping variant of this rule, the typing contexts remain inert.

The ST-Met rule is a counterpart of a subtyping rule for function types in DOT, which plays a major role in DOT being conjectured to be undecidable [9]. Correspondingly, in roDOT, this rule makes it difficult to test whether a particular type is read-write or read-only.

The ST-Sel* rules give bounds to type selections. They substitute the provided argument for the receiver parameter.

The ST-N-M rule makes \perp the greatest lower bound of N and $\{M : \perp..\perp\}$, expressing that nothing can be both read-write and read-only. The other ST-N-* rules establish N as a lower bound of read-only types. As in the baseline DOT, there is no subtyping between different recursive types.

$$\begin{array}{c}
\overline{\Gamma; \rho \vdash T <: T} \text{ (ST-Refl)} \\
\frac{\Gamma; \rho \vdash T_1 <: T_2 \quad \Gamma; \rho \vdash T_2 <: T_3}{\Gamma; \rho \vdash T_1 <: T_3} \text{ (ST-Trans)} \\
\overline{\Gamma; \rho \vdash T <: \top} \text{ (ST-Top)} \\
\overline{\Gamma; \rho \vdash \perp <: T} \text{ (ST-Bot)} \\
\frac{\rho \vdash T_1 \approx T_2}{\Gamma; \rho \vdash T_1 <: T_2} \text{ (ST-Eq)} \\
\overline{\Gamma; \rho \vdash T_1 <: T_1 \vee T_2} \text{ (ST-Or1)} \\
\overline{\Gamma; \rho \vdash T_2 <: T_1 \vee T_2} \text{ (ST-Or2)} \\
\frac{\Gamma; \rho \vdash T_1 <: T_3 \quad \Gamma; \rho \vdash T_2 <: T_3}{\Gamma; \rho \vdash T_1 \vee T_2 <: T_3} \text{ (ST-Or)} \\
\frac{\Gamma; \rho \vdash T_1 <: T_2 \quad \Gamma; \rho \vdash T_1 <: T_3}{\Gamma; \rho \vdash T_1 <: T_2 \wedge T_3} \text{ (ST-And)} \\
\overline{\Gamma; \rho \vdash T <: T} \text{ (ST-Refl)} \\
\overline{\Gamma; \rho \vdash T_1 \wedge T_2 <: T_1} \text{ (ST-And1)} \\
\overline{\Gamma; \rho \vdash T_1 \wedge T_2 <: T_2} \text{ (ST-And2)} \\
\frac{\Gamma; \rho \vdash x : \{B(r) : T_1..T_2\}}{\Gamma; \rho \vdash [x_2/r]T_1 <: x.B(x_2)} \text{ (ST-SelL)} \\
\frac{\Gamma; \rho \vdash x : \{B(r) : T_1..T_2\}}{\Gamma; \rho \vdash x.B(x_2) <: [x_2/r]T_2} \text{ (ST-SelU)} \\
\frac{\Gamma; \rho \vdash T_3 <: T_1 \quad \Gamma; \rho \vdash T_2 <: T_4}{\Gamma; \rho \vdash \{B(r) : T_1..T_2\} <: \{B(r) : T_3..T_4\}} \text{ (ST-Typ)} \\
\frac{\Gamma; \rho \vdash T_3 <: T_1 \quad \Gamma; \rho \vdash T_2 <: T_4}{\Gamma; \rho \vdash \{a : T_1..T_2\} <: \{a : T_3..T_4\}} \text{ (ST-Fld)} \\
\overline{\Gamma; \rho \vdash \mathbf{N} \wedge \{M(r) : \perp.. \perp\} <: \perp} \text{ (ST-N-M)} \\
\overline{\Gamma; \rho \vdash \mathbf{N} <: \mu(s : T)} \text{ (ST-N-Rec)} \\
\overline{\Gamma; \rho \vdash \mathbf{N} <: \{a : T_1..T_2\}} \text{ (ST-N-Fld)} \\
\overline{\Gamma; \rho \vdash \mathbf{N} <: \{A(r) : T_1..T_2\}} \text{ (ST-N-Typ)} \\
\frac{\Gamma; \rho \vdash T_3 <: T_1 \quad \Gamma, z : T_3; \rho \vdash T_6 <: T_5 \quad \Gamma, z : T_3, r : T_6; \rho \vdash T_2 <: T_4}{\Gamma; \rho \vdash \{m(z : T_1, r : T_5) : T_2\} <: \{m(z : T_3, r : T_6) : T_4\}} \text{ (ST-Met)} \\
\overline{\Gamma; \rho \vdash \mathbf{N} <: \{m(z : T_1, r : T_3) : T_2\}} \text{ (ST-N-Met)} \\
\overline{\Gamma; \rho \vdash \{B(r) : T_1..T_2\} \wedge \{B(r) : T_3..T_4\} <: \{B(r) : T_1 \vee T_3..T_2 \wedge T_4\}} \text{ (ST-TypAnd)} \\
\overline{\Gamma; \rho \vdash T_1 \wedge (T_2 \vee T_3) <: (T_1 \wedge T_2) \vee (T_1 \wedge T_3)} \text{ (ST-Dist)}
\end{array}$$

■ **Figure 4** Subtyping.

4.2.2 Variable Typing Rules

The VT-Var rule gives variables the type assigned by the typing context, VT-Sub adds subsumption and VT-AndI gives variables intersection types.

The typing rules VT-RecE and VT-RecI allow opening and closing recursive types. Both rules require that the inner type T is independent of the mutability $s.M$ of s , written T **indep** s . The introduction rule additionally requires the inner type to be read-only by requiring that the **ro** operation does not change the original type.

To ensure that the type selection $x.M$ is valid for every variable x , we add the axiom VT-MutTop, which gives every variable the type $\{M : \perp.. \top\}$.

4.2.3 Term Typing Rules

Typing of terms requires that all variables occurring free in a term, but not as a part of a type (we call them t-free), are visible, as discussed in Section 3.3. For each such occurrence, there is a premise $\Gamma \mathbf{vis} x$ in the corresponding rule. By Vis-Var, only the variables after the ! are visible for term typing. Variables in the context before the ! can still be used for type selection. This separation makes use of our explicit notation for a variable used as a term, written vx , and typed using only the part of the context after the !. A plain variable x that appears in a type selection $x.A$ is typed using the full typing context.

The TT-Var rule gives types given by variable typing to visible variables. TT-Sub adds subsumption for term types. TT-Let types let terms as in the baseline DOT.

$$\begin{array}{c}
\frac{! \notin \Gamma_2}{\Gamma_1, x : T, \Gamma_2 \mathbf{vis} x} \text{(Vis-Var)} \qquad \frac{\Gamma; \rho \vdash x : \mu(s : T)}{T \mathbf{indep} s} \text{(VT-RecE)} \\
\frac{\Gamma = \Gamma_1, x : T, \Gamma_2}{\Gamma; \rho \vdash x : T} \text{(VT-Var)} \qquad \frac{\Gamma; \rho \vdash x : [x/s]T}{\Gamma; \rho \vdash x : [x/s]T} \text{(VT-RecI)} \\
\frac{\Gamma; \rho \vdash x : T_1 \quad \Gamma; \rho \vdash T_1 <: T_2}{\Gamma; \rho \vdash x : T_2} \text{(VT-Sub)} \qquad \frac{\Gamma; \rho \vdash x : [x/s]T \quad \Gamma \mathbf{vis} x}{\Gamma; \rho \vdash x : \mu(s : T)} \text{(VT-RecI)} \\
\frac{\Gamma; \rho \vdash x : T_1 \quad \Gamma; \rho \vdash x : T_2}{\Gamma; \rho \vdash x : T_1 \wedge T_2} \text{(VT-AndI)} \qquad \frac{\Gamma; \rho \vdash x : T}{\Gamma; \rho \vdash x : \{M(r_0) : \perp.. \top\}} \text{(VT-MutTop)} \\
\frac{\Gamma; \rho \vdash x : T \quad \Gamma \mathbf{vis} x}{\Gamma; \rho \vdash vx : T} \text{(TT-Var)} \qquad \frac{\Gamma; \rho \vdash x_1 : \{m(z : T_1, r : T_3) : T_2\} \quad \Gamma; \rho \vdash x_2 : T_1 \quad \Gamma \mathbf{vis} x_2}{\Gamma; \rho \vdash x_1 : [x_2/z]T_3 \quad \Gamma \mathbf{vis} x_1} \text{(TT-Apply)} \\
\frac{\Gamma; \rho \vdash t : T_1 \quad \Gamma; \rho \vdash T_1 <: T_2}{\Gamma; \rho \vdash t : T_2} \text{(TT-Sub)} \qquad \frac{\Gamma; \rho \vdash x_1 : T_1 \quad \Gamma \mathbf{vis} x_1}{\Gamma; \rho \vdash x : \{M(r) : \perp.. \perp\}} \text{(TT-Write)} \\
\frac{\Gamma; \rho \vdash t_1 : T_1 \quad z \notin \text{fv } T_2 \quad \Gamma, z : T_1; \rho \vdash t_2 : T_2}{\Gamma; \rho \vdash \text{let } z = t_1 \text{ in } t_2 : T_2} \text{(TT-Let)} \qquad \frac{\Gamma, s : T_1; \rho \vdash d : T_1 \quad z \notin \text{fv } T_2 \quad T_1 \mathbf{indep} s \quad \Gamma, z : \mu(s : T_1) \wedge \{M(r) : \perp.. \perp\}; \rho \vdash t : T_2}{\Gamma; \rho \vdash \text{let } z = \nu(s : T_1)d \text{ in } t : T_2} \text{(TT-New)} \\
\frac{\Gamma; \rho \vdash x : \{a : T_1..T_2\} \quad \Gamma \mathbf{vis} x \quad \Gamma; \rho \vdash T_2 \mathbf{ro} T_3 \quad \Gamma; \rho \vdash T_2 \mathbf{mu}(r) T_4}{\Gamma; \rho \vdash x.a : T_3 \wedge \{M(r) : \perp..(T_4 \vee x.M(r))\}} \text{(TT-Read)}
\end{array}$$

■ Figure 5 Typing.

$$\begin{array}{c}
\overline{\Gamma, s : T_4; \rho \vdash \{A(r) = T\} : \{A(r) : T..T\}} \text{(DT-Typ)} \\
\overline{\Gamma, s : T_4; \rho \vdash \{A(r) = T\} : \{A(r) : \perp..T\}} \text{(DT-TypB)} \\
\frac{\Gamma, s : T_4; \rho \vdash x : T \quad \Gamma, s : T_4 \text{ vis } x}{\Gamma, s : T_4; \rho \vdash \{a = x\} : \{a : T..T\}} \text{(DT-Fld)} \\
\frac{\Gamma, s : T_4; \rho \vdash d_1 : T_1 \quad \Gamma, s : T_4; \rho \vdash d_2 : T_2 \\ d_1 \text{ and } d_2 \text{ have distinct member names}}{\Gamma, s : T_4; \rho \vdash d_1 \wedge d_2 : T_1 \wedge T_2} \text{(DT-And)} \\
\frac{z \notin \text{fv } T_1 \cup \text{fv } T_4, r \notin \text{fv } T_3 \cup \text{fv } T_1 \cup \text{fv } T_4 \\ \Gamma, s : T_4, !, z : T_1, r : T_4 \wedge [r/s]T_4 \wedge T_3; \rho \vdash t : T_2}{\Gamma, s : T_4; \rho \vdash \{m(z, r) = t\} : \{m(z : T_1, r : T_3) : T_2\}} \text{(DT-Met)}
\end{array}$$

■ **Figure 6** Definition typing.

The TT-New rule should give a read-write type to every constructed object. Therefore, the type of z in the context for typing t in $\text{let } z = \nu(s : T_1) \text{ in } t$ is changed to $\text{rw}(\mu(s : T_1))$. The type T_1 written for s must correspond to the definitions and cannot refer to $s.M$. Because objects are given a recursive type, this corresponds to the requirement that recursive types must always be read-only.

The TT-Apply rule for method calls substitutes both the parameter and the receiver into the result type. The type declared for r restricts the type of the receiver. The typing rule for method application checks that both the receiver and the argument have the expected type. For example, if the receiver parameter type is declared to have a read-write type, the method can be called only on read-write references.

In TT-Write, we add a premise to ensure that the reference whose field we are mutating is read-write: $x : \{M : \perp.. \perp\}$.

Finally, in TT-Read, we need to viewpoint-adapt the type T_2 of the field with the mutability of the reference x through which we are reading the field. For $x : \{a : T_1..T_2\}$, we change the type of $x.a$ from T_2 to T_5 , and add a premise $\Gamma \vdash x \triangleright T_2 \rightarrow T_5$.

Viewpoint Adaptation

In Section 3.3, we described how viewpoint adaptation can be expressed in terms of two new type relations $\Gamma \vdash T \mathbf{ro} T_r$ and $\Gamma \vdash T \mathbf{mu} T_m$. The relations are defined together in Figure 7. The operation $\Gamma \vdash T \mathbf{ro} T_r$ means that T_r is a supertype of T that is definitely read-only. The **ro** relation extracts the parts of a type other than mutability. Thus, the relation maps the mutability type member type $\{M : T..T'\}$ to \top . The relation is the identity on the \top type, field and method declarations, and type declarations other than M . Because we want $T <: T_r$ whenever $\Gamma \vdash T \mathbf{ro} T_r$, the relation must also be the identity on recursive object types, because they do not participate in any subtyping relationships other than reflexivity and subtyping with \perp and \top . To enforce this, the typing rule for recursion introduction needs to ensure that the self type T is read-only.

$$\begin{array}{c}
\frac{}{\Gamma; \rho \vdash \top \mathbf{ro} \top} \text{(TS-Top)} \\
\Gamma; \rho \vdash \top \mathbf{mu}(r) \top \\
\\
\frac{}{\Gamma; \rho \vdash \perp \mathbf{ro} \mathbf{N}} \text{(TS-Bot)} \\
\Gamma; \rho \vdash \perp \mathbf{mu}(r) \perp \\
\\
\frac{T = \{A(r) : T_1..T_2\}}{\Gamma; \rho \vdash T \mathbf{ro} T} \text{(TS-Typ)} \\
\Gamma; \rho \vdash T \mathbf{mu}(r_0) \top \\
\\
\frac{T = \{m(z : T_1, r : T_3) : T_2\}}{\Gamma; \rho \vdash T \mathbf{ro} T} \text{(TS-Met)} \\
\Gamma; \rho \vdash T \mathbf{mu}(r_0) \top \\
\\
\frac{T = \{a : T_1..T_2\}}{\Gamma; \rho \vdash T \mathbf{ro} T} \text{(TS-Fld)} \\
\Gamma; \rho \vdash T \mathbf{mu}(r) \top \\
\\
\frac{\Gamma; \rho \vdash x : \{B(r) : T_1..T_2\} \\
\Gamma; \rho \vdash [x_2/r]T_2 \mathbf{ro} T_3 \\
\Gamma; \rho \vdash [x_2/r]T_2 \mathbf{mu}(r_0) T_4}{\Gamma; \rho \vdash x.B(x_2) \mathbf{ro} T_3} \text{(TS-Sel)} \\
\Gamma; \rho \vdash x.B(x_2) \mathbf{mu}(r_0) T_4 \\
\\
\frac{}{\Gamma; \rho \vdash \{M(r) : T_1..T_2\} \mathbf{ro} \top} \text{(TS-M)} \\
\Gamma; \rho \vdash \{M(r) : T_1..T_2\} \mathbf{mu}(r) T_2 \\
\\
\frac{T = \mu(s : T_1)}{\Gamma; \rho \vdash T \mathbf{ro} T} \text{(TS-Rec)} \\
\Gamma; \rho \vdash T \mathbf{mu}(r) \top \\
\\
\frac{\Gamma; \rho \vdash T_1 \mathbf{ro} T_2 \\
\Gamma; \rho \vdash T_3 \mathbf{ro} T_4}{\Gamma; \rho \vdash T_1 \wedge T_3 \mathbf{ro} T_2 \wedge T_4} \text{(TS-AndR)} \\
\\
\frac{\Gamma; \rho \vdash T_1 \mathbf{mu}(r) T_2 \\
\Gamma; \rho \vdash T_3 \mathbf{mu}(r) T_4}{\Gamma; \rho \vdash T_1 \wedge T_3 \mathbf{mu}(r) T_2 \wedge T_4} \text{(TS-AndM)} \\
\\
\frac{\Gamma; \rho \vdash T_1 \mathbf{ro} T_2 \\
\Gamma; \rho \vdash T_3 \mathbf{ro} T_4}{\Gamma; \rho \vdash T_1 \vee T_3 \mathbf{ro} T_2 \vee T_4} \text{(TS-OrR)} \\
\\
\frac{\Gamma; \rho \vdash T_1 \mathbf{mu}(r) T_2 \\
\Gamma; \rho \vdash T_3 \mathbf{mu}(r) T_4}{\Gamma; \rho \vdash T_1 \vee T_3 \mathbf{mu}(r) T_2 \vee T_4} \text{(TS-OrM)}
\end{array}$$

■ **Figure 7** Splitting relations.

On intersection and union types, the **ro** relation is defined recursively on the two parts of the type. For a type selection $x.B(x_2)$, **ro** is applied recursively to the upper bound of B in the type of x . For the bottom type \perp , **ro** cannot simply return \perp itself because \perp is read-write since $\perp <: \{M : \perp.. \perp\}$. We make \mathbf{N} a subtype only of types that are definitely known to be read-only, including declaration types other than \mathbf{M} and all recursive types.

The **mu** relation is defined to return T_2 for $\{M(r) : \perp..T_2\}$, to recurse on intersection and union types and into the upper bound of a type selection, and to return \top for all other (read-only) types. Because in TS-M, the type T_2 may refer to the receiver r , the **mu** relation is parameterized by a variable that binds to this receiver. This variable is used in the declaration of \mathbf{M} in the viewpoint-adapted type in TT-Read.

4.2.4 Definition Typing

Definition typing, shown in Figure 6 (DT-*), is only used in the context of the TT-New rule, where the self reference s is the last variable in the typing context. Singling out this variable from the rest of the typing context is important for the DT-Met rule, in order to give r a type derived from the type of the object.

Typing of field definitions DT-Fld allows using s as the value of the field. It requires the value of the field to be visible, and gives the field a type with tight bounds. Typing type members allows tight bounds, but we also allow fixing just the upper bound and leaving the lower bound to be \perp . This allows declarations of ordinary type members to be similar to the declarations of the mutability member \mathbf{M} , which always have \perp as the lower bound.

In DT-Met, z is given the parameter type specified in the method declaration, but the type for r is formed by intersecting the declared type with the type T_4 given to s in TT-New. The rule looks up the type of s in the context and gives the same type to r . Additionally, a version of T_4 with s replaced by r is added to the intersection, allowing deriving the recursive type $\mu(s : T_4)$ for r .

Variables other than the parameter and the receiver are hidden from the context and not allowed to be used as a value in the method. That is achieved in DT-Met by splitting the typing context for the method body into two parts separated with an ! symbol.

4.3 Runtime Configuration

The syntax of runtime configurations is shown in Figure 8. A machine configuration c consists of a focus of execution t , stack s , runtime environment ρ and heap Σ . Each frame of the stack is a let term with a hole \square into which the reduced focus of execution will be substituted. The runtime environment ρ is a new part of a configuration, which maps references w to the locations y to which they refer.

Because the only items in the heap are objects, we omit the header $\nu(s : R)$ and store only the definition d , which is an intersection of field, method and type member definitions. The values of fields of heap objects are restricted to only locations y by heap correspondence. Since each object in the heap is at a known location y , we substitute this location y for any occurrences of the self variable s in the member definitions.

Valid configurations are given a type under an inert context F . The rules for typing configurations are given in Figure 9. Stack typing assigns to each stack a pair of types, an input type T_1 and an output type T_3 , indicating that if the focus of execution reduces to a value of type T_1 , then the entire stack will reduce to a value of type T_3 . The environment ρ must correspond to the typing context, meaning that each reference w corresponding to a location y under ρ must appear after y in F and have the same type except for mutability. The heap must correspond to F , which requires that for every location y in F , an object has to exist on the heap, and the object must have the correct type with y substituted for s . Finally, to type a configuration, the CT-Corr rule checks environment correspondence and heap correspondence, then types the focus of execution t and the stack σ , checks that the type of the focus of execution matches the input type of the stack, and finally gives the output type of the stack to the entire configuration.

$\Sigma ::=$	Heap	$Q ::=$	Member type
\cdot	empty heap	$\{a : T..T\}$	tight field
$\Sigma, y \rightarrow d$	heap object	$\{m(z : T_1, r : T_3) : T_2\}$	method
$\sigma ::=$	Stack	$\{A(r) : T..T\}$	tight type
\cdot	empty stack	$\{A(r) : \perp..T\}$	upper-bounded type
let $z = \square$ in $t :: \sigma$	let frame	$R ::=$	Record type
$\rho ::=$	Environment	Q	member
\cdot	empty environment	$R_1 \wedge R_2$	intersection
$\rho, w \rightarrow y$	assignment	$S ::=$	Inert type
$c ::=$	Configuration	$\mu(s : R) \wedge \{M(r) : \perp..T\}$	object
$\langle t; \sigma; \rho; \Sigma \rangle$		$F ::=$	Inert context
$\Gamma_h ::=$	Heap Context		empty
$\Gamma, y/s : R$		$F, y : S$	binding

■ **Figure 8** Runtime.

$$\begin{array}{c}
\frac{F; \rho \vdash T_1 <: T_2}{F; \rho \vdash \cdot : T_1, T_2} \text{(CT-EmptyS)} \quad \frac{F, z : T_1; \rho \vdash t : T_2 \quad z \notin \text{fv } T_2 \quad F; \rho \vdash \sigma : T_2, T_3}{F; \rho \vdash \text{let } z = \square \text{ in } t :: \sigma : T_1, T_3} \text{(CT-LetS)} \\
\frac{}{F; \rho \vdash \sim \cdot} \text{(CT-EmptyH)} \quad \frac{F_1; \rho \vdash F_2 \sim \Sigma}{F_1; \rho \vdash F_2, w : T \sim \Sigma} \text{(CT-RefH)} \\
\frac{}{\Gamma \sim \cdot} \text{(CT-EmptyE)} \\
\frac{F_1, y/s : R; \rho \vdash d : [y/s]R \quad F_1; \rho \vdash F_2 \sim \Sigma \quad R \text{ indep } s}{F_1; \rho \vdash F_2, y : \mu(s : R) \wedge \{M(r) : \perp.. \perp\} \sim \Sigma, y \rightarrow d} \text{(CT-ObjH)} \\
\frac{\Gamma = \Gamma_1, w : \mu(s : R) \wedge \{M(r) : \perp.. T\}, \Gamma_2 \quad \Gamma_1 \sim \rho \quad \Gamma_1 = \Gamma_3, y : \mu(s : R) \wedge \{M(r) : \perp.. \perp\}, \Gamma_4}{\Gamma \sim \rho, w \rightarrow y} \text{(CT-RefE)} \\
\frac{F; \rho \vdash F \sim \Sigma \quad \text{all fields in } \Sigma \text{ are locations}}{F; \rho \sim \Sigma} \text{(CT-CorrH)} \quad \frac{F \sim \rho \quad F; \rho \sim \Sigma \quad F; \rho \vdash t : T_1 \quad F; \rho \vdash \sigma : T_1, T_2 \quad \text{no locations in } t \text{ and } \sigma}{F \vdash \langle t; \sigma; \rho; \Sigma \rangle : T_2} \text{(CT-Corr)} \\
\frac{z \notin \text{fv } T_1 \cup \text{fv } T_4, r \notin \text{fv } T_3 \cup \text{fv } T_1 \cup \text{fv } T_4 \quad \Gamma, !, z : T_1, r : [y/s]T_4 \wedge [r/s]T_4 \wedge T_3; \rho \vdash t : T_2}{\Gamma, y/s : T_4; \rho \vdash \{m(z, r) = t\} : \{m(z : T_1, r : T_3) : T_2\}} \text{(HT-Met)}
\end{array}$$

■ **Figure 9** Configuration typing and correspondence.

4.3.1 Environment

When a new object is created, both a fresh location y and a fresh reference w are created, with the same read-write type. The location y is put on the heap, the reference w is put into the focus of execution, and w is connected to y by the environment ρ . When writing a reference w to a field, the corresponding location y is stored on the heap. Its mutability is determined by the type of the field. When reading the value of field a from a reference w_1 , a new reference w_2 is created for the location y_2 stored in the field of the object stored at location $y_1 = \rho(w_1)$ on the heap. The new reference w_2 is given the type of y_2 with the mutability changed by viewpoint adaptation to be an upper bound of the mutability of the field and of the reference w_1 .

4.3.2 Heap Correspondence

The heap correspondence relation checks that the type of each location y in the typing context corresponds to the object stored at y in the heap.

The type of y in the context is the read-write version of the type specified when creating the object. That is, when a literal $\nu(s : T)d$ leads to creating y on the heap, then $\Sigma(y) = [y/s]d$ and $F(y) = \mu(s : T) \wedge \{M : \perp.. \perp\}$.

To check that the definition of the object corresponds to its type, we define a modified definition typing. The baseline DOT uses the same rules for typing object literals in let statements and typing heap items in heap correspondence. In roDOT, to preserve typing

after the substitution, the type of r in the context for method bodies must be changed to use y instead of s in T_4 . Because of that, we have a set of definition typing rules for heap items HT-*, similar to the set of definition typing rules DT-* in Figure 6. They give types to definitions on the heap in a *heap context* with the special syntax $\Gamma, y/s : T_4$. We show only the HT-Met rule which differs from DT-Met in that it removes s from the typing context and substitutes y for s in T_4 . The reason for this is so that in the HT-Met rule shown in Figure 6, r can be given the types $[y/s]T_4$ and $[r/s]T_4$. Other HT-* rules not shown here are similar to the DT-* rules, except that HT-Fld does not put s into the context for typing x .

4.4 Reduction

Reduction is defined in Figure 10. There is a reduction step for each kind of term, which produces the next configuration. We change the reduction from the baseline DOT to use reference variables w to represent references in runtime configurations, rather than directly using the locations y . Rules that access the heap have additional premises that relate the references with the corresponding locations in the environment. If the term is a single variable and the stack is empty, then no step can be taken and the evaluation ends in a final configuration. The R-LetPush and R-LetLoc rules work with the stack in the same way as in the baseline DOT. The R-Write rule overwrites the value of a field on the heap. In a R-Read step, a new reference to an object is created in the focus for a location that was stored in a field. The R-Apply rule is changed to apply a method of an object instead of a function value. It substitutes both the parameter and the receiver into the method body and proceeds to reduce it. In a R-LetNew step, the heap is extended with a new object y and the environment is extended with a new reference w with the same read-write type. The definition of the object on the heap is constructed from the provided object literal by replacing all references by corresponding locations and replacing the self variable by y .

$$\begin{array}{c}
\frac{y_1 \rightarrow \dots_1 \{a = \underline{y_2}\} \dots_2 \in \Sigma \quad w_1 \rightarrow y_1 \in \rho_1 \quad \rho_2 = \rho_1, w_2 \rightarrow y_2 \quad (w_2 \text{ fresh})}{\langle w_1.a; \sigma; \rho_1; \Sigma \rangle \mapsto \langle \nu w_2; \sigma; \rho_2; \Sigma \rangle} \text{(R-Read)} \\
\\
\frac{w_1 \rightarrow y_1 \in \rho \quad y_1 \rightarrow \dots_1 \{a = \underline{y_2}\} \dots_2 \in \Sigma_1 \quad w_3 \rightarrow y_3 \in \rho \quad \Sigma_2 = \Sigma_1[y_1 \rightarrow \dots_1 \{a = y_3\} \dots_2]}{\langle w_1.a := w_3; \sigma; \rho; \Sigma_1 \rangle \mapsto \langle \nu w_3; \sigma; \rho; \Sigma_2 \rangle} \text{(R-Write)} \\
\\
\frac{w_1 \rightarrow y_1 \in \rho \quad y_1 \rightarrow \dots_1 \{m(z, r) = t\} \dots_2 \in \Sigma}{\langle w_1.m w_2; \sigma; \rho; \Sigma \rangle \mapsto \langle [w_1/r][w_2/z]t; \sigma; \rho; \Sigma \rangle} \text{(R-Apply)} \\
\\
\frac{\rho_2 = \rho_1, w \rightarrow y \quad \Sigma_2 = \Sigma_1, y \rightarrow [y/s][\rho_1]d \quad (y, w \text{ fresh})}{\langle \text{let } z = \nu(s : T)d \text{ in } t; \sigma; \rho_1; \Sigma_1 \rangle \mapsto \langle [w/z]t; \sigma; \rho_2; \Sigma_2 \rangle} \text{(R-LetNew)} \\
\\
\frac{}{\langle \text{let } z = t_1 \text{ in } t_2; \sigma; \rho; \Sigma \rangle \mapsto \langle t_1; \text{let } z = \square \text{ in } t_2 :: \sigma; \rho; \Sigma \rangle} \text{(R-LetPush)} \\
\\
\frac{}{\langle \nu w; \text{let } z = \square \text{ in } t :: \sigma; \rho; \Sigma \rangle \mapsto \langle [w/z]t; \sigma; \rho; \Sigma \rangle} \text{(R-LetLoc)}
\end{array}$$

■ **Figure 10** Reduction.

$$\begin{array}{c}
\text{F} \vdash \langle t; \sigma; \rho; \Sigma \rangle \text{ mreach } y_1 \\
y_1 \rightarrow \dots_1 \{a = y_2\} \dots_2 \in \Sigma \\
\hline
\text{F}; \rho \vdash y_1 : \{a : \perp.. \{M(r) : \perp.. \perp\}\} \\
\text{F} \vdash \langle t; \sigma; \rho; \Sigma \rangle \text{ mreach } y_2 \quad (\text{Rea-Fld})
\end{array}
\qquad
\begin{array}{c}
t \text{ tfree } w \vee \sigma \text{ tfree } w \\
w \rightarrow y \in \rho \\
\hline
\text{F}; \rho \vdash w : \{M(r) : \perp.. \perp\} \\
\text{F} \vdash \langle t; \sigma; \rho; \Sigma \rangle \text{ mreach } y \quad (\text{Rea-Term})
\end{array}$$

■ **Figure 11** Mutable reachability.

4.5 Example

In roDOT, we can rewrite the example from Section 3.2 with the intended mutability types.

Assume that T is a read-only type in the sense that $\Gamma \vdash T \text{ ro } T$. Then we can let the field have a read-write type by adding the mutability marker. By using the mutability marker as the type of r in m_s , we can express that the setter can only be called on read-write references. By adding $\{M(r_0) : \perp..r.M\}$ to the result type of m_g , we ensure that the getter only returns a read-write reference if called on a read-write reference. The type of the object will be $T_{o1} \triangleq \mu(s : \{a : T \wedge \{M : \perp.. \perp\}\} \wedge \{m_s(z : T \wedge \{M : \perp.. \perp\}, r : \{M : \perp.. \perp\}) : \top\} \wedge \{m_g(z : \top, r : \top) : T \wedge \{M(r_0) : \perp..r.M(r_0)\}\})$. Given an initial value x of type $T \wedge \{M : \perp.. \perp\}$, it can be instantiated in $\text{let } z = \nu(s : T_{o1})\{a = x\} \wedge \{m_s(r, z) = r.a := z\} \wedge \{m_g(r, z) = r.a\}$ in t .

To allow storing read-only values in the a field as well, we can parameterize the mutability of the field using a type member A . The type of the object will then be: $T_{o2} \triangleq \mu(s : \{A : \perp.. \top\} \wedge \{a : T \wedge \{M(r_0) : \perp..s.A(r_0)\}\} \wedge \{m_s(z : T \wedge \{M(r_0) : \perp..s.A(r_0)\}, r : \{M : \perp.. \perp\}) : \top\} \wedge \{m_g(z : \top, r : \top) : T \wedge \{M(r_0) : \perp..r.M(r_0) \vee s.A(r)\}\})$.

5 Properties and their Proofs

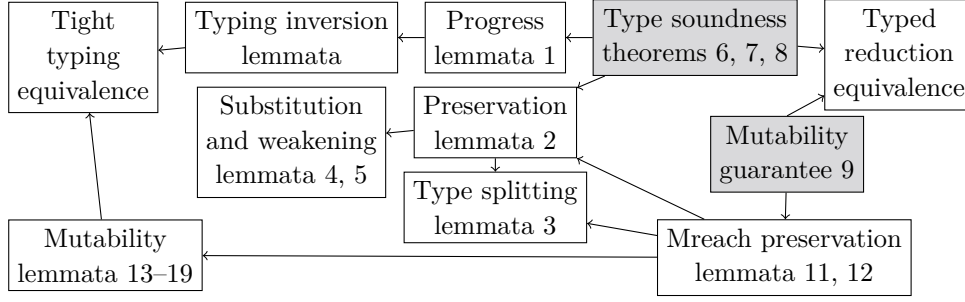
In this section we state and prove type soundness of roDOT and the immutability guarantee.

5.1 Immutability Guarantee

We define the immutability guarantee that roDOT provides as follows: an object on the heap Σ in a configuration $c = \langle t; \sigma; \rho; \Sigma \rangle$ typed under F can be modified in an execution starting from c only if it is *mutably reachable*, i.e., reachable from the configuration using only read-write references. Mutable reachability is formally defined in Figure 11. By the Rea-Term rule, y is mutably reachable if a read-write reference to it occurs in the focus of execution t or the stack σ in a position other than in type selection (the reference is t-free in t or σ). By the Rea-Fld rule, y_2 is mutably reachable if its location is stored in the field of some mutably reachable object y_1 and this field has read-write type. The immutability guarantee does not say anything about objects which are yet to be created; these may be modified. We will prove that if an object y on the heap Σ in the initial configuration c is not mutably reachable, then it will not appear on the left-hand side of a write term in any execution starting from c .

5.2 Proofs

Figure 12 summarizes the overall structure of groups of properties that we have proved about the type system and their major dependencies. The full statements of all lemmata and their proofs are in a supplementary technical report [6]. We discuss the most significant properties in the rest of this section; they are identified by numbers in Figure 12.



■ **Figure 12** Overview of properties and dependencies within proofs of the main theorems.

$$\begin{array}{c}
 t_0 = w_1.a \quad \mathbb{F} \vdash \langle t_0; \sigma_1; \rho_1; \Sigma_1 \rangle : T_0 \quad \mathbb{F}; \rho_1 \vdash w_1 : \{a : T_4..T_3\} \quad w_1 \rightarrow y_1 \in \rho_1 \\
 y_1 \rightarrow \dots \rightarrow \{a = y_2\} \dots \in \Sigma_1 \quad \mathbb{F}; \rho_1 \vdash T_3 \mathbf{mu}(r) T_7 \\
 \mathbb{F} = \mathbb{F}_3, y_2 : \mu(s_1 : R_1) \wedge \{\mathbf{M}(r_0) : \perp.. \perp\}, \mathbb{F}_4 \\
 T_2 = \mu(s_1 : R_1) \wedge \{\mathbf{M}(r) : \perp..(T_7 \vee w_1.\mathbf{M}(r))\} \quad \rho_2 = \rho_1, w_2 \rightarrow y_2 \\
 \hline
 \mathbb{F} \vdash \langle t_0; \sigma_1; \rho_1; \Sigma_1 \rangle : T_0 \mapsto \mathbb{F}, w_2 : T_2 \vdash \langle w_2; \sigma_1; \rho_2; \Sigma_1 \rangle \quad (\text{TR-Read})
 \end{array}$$

$$\begin{array}{c}
 t_0 = \text{let } z = \nu(s : R)d \text{ in } t \quad \mathbb{F} \vdash \langle t_0; \sigma_1; \rho_1; \Sigma_1 \rangle : T_0 \quad \rho_2 = \rho_1, w_1 \rightarrow y_1 \\
 T = \mu(s : R) \wedge \{\mathbf{M}(r_0) : \perp.. \perp\} \quad \Sigma_2 = \Sigma_1, y_1 \rightarrow [y_1/s][\rho_1]d \\
 \hline
 \mathbb{F} \vdash \langle t_0; \sigma_1; \rho_1; \Sigma_1 \rangle : T_0 \mapsto \mathbb{F}, y_1 : T, w_1 : T \vdash \langle [w_1/z]t; \sigma_1; \rho_2; \Sigma_2 \rangle \quad (\text{TR-LetNew})
 \end{array}$$

■ **Figure 13** Typed reduction.

5.2.1 Proof of Type Soundness

Type soundness ensures that evaluation of a typed term does not get stuck in a non-final configuration where no reduction rule can be applied. Similarly to kDOT, it is shown using two properties of reduction: Progress means that unless a typed configuration is final, a step can be taken. Preservation means that the step retains the type of the configuration.

We state the properties differently than kDOT. Typing a configuration requires a typing context, which after taking a step such as creating a new object, might have to be extended to give a type to the newly created location. The usual reduction rules do not specify how the typing context should change. For the proofs, we define a typed variant of reduction. It transforms configurations in the same way as the rules in Figure 10. Additionally, it requires the configuration to have a type, and also produces a typing context for the next configuration. This makes type preservation easier to state because the typing context is fixed, and makes it possible to state a similar preservation property for proving the immutability guarantee.

The typed reduction rules for the two interesting cases are shown in Figure 13. In TR-Read, a type for a new reference variable is constructed. This type must be precise enough so that the resulting term keeps the expected type, but at the same time, it must be read-only if either the field or the reference to the containing object was read-only. We construct the type by taking the recursive part of the heap type of the object and changing the mutability. The new mutability is an upper bound of the mutability of the containing object, expressed by $w_1.\mathbf{M}$, and the mutability of the field type given by \mathbf{mu} .

In TR-LetNew, both the new location y_1 and the reference w_1 are given a read-write type based on the object literal. The other typed reduction rules (not shown) are straightforward because the typing context does not change.

We state progress and preservation for each of these 6 typed reduction rules, which each handle one syntactic form of a term. Progress states that if a configuration with such a term in the focus of execution has a type, then a typed rule can be applied. Preservation states that the context produced by the rule gives the new configuration the same type. Lemmata 1 and 2 are examples of progress and preservation for the TR-Read rule.

► **Lemma 1 (PgRead).** *If $F \vdash \langle w_1.a; \sigma_1; \rho_1; \Sigma_1 \rangle : T_0$, then there exist w_2, T_2 and ρ_2 , such that $F \vdash \langle w_1.a; \sigma_1; \rho_1; \Sigma_1 \rangle : T_0 \mapsto F, w_2 : T_2 \vdash \langle vw_2; \sigma_1; \rho_2; \Sigma_1 \rangle$.*

► **Lemma 2 (TPRead).** *If $F \vdash \langle w_1.a; \sigma_1; \rho_1; \Sigma_1 \rangle : T_0 \mapsto F, w_2 : T_2 \vdash \langle vw_2; \sigma_1; \rho_2; \Sigma_1 \rangle$, then $F, w_2 : T_2 \vdash \langle vw_2; \sigma_1; \rho_2; \Sigma_1 \rangle : T_0$.*

Note that these lemmata also imply progress and preservation of the untyped reduction rules. For progress, since the premises of each typed reduction rule contain all the premises of the corresponding untyped reduction rule, if progress ensures that some typed reduction rule applies to a configuration, then the corresponding untyped reduction rule also applies. Preservation for untyped reduction rules requires that there exist some extended typing context in which the next configuration has the same type, and the typed reduction rules explicitly provide this context.

The proofs of these lemmata follow the recipe from [12] and [14]. We define precise, tight and invertible variants of typing for variables, and a tight variant of subtyping. Invertible typing together with heap correspondence ensures that objects used in Read, Write or Apply terms have the member needed for progress, and preservation. In an inert context, typing and invertible typing are equivalent.

Notable differences from the baseline DOT are in the TR-Read and TR-LetNew cases.

In Lemma 2, it must be shown that the new reference variable w_2 has the expected viewpoint-adapted type as defined by the TT-Read typing rule. This type T_2 is formed by an intersection of a read-only part and a mutability member declaration. To show that the reference has the read-only part of the type, we use Lemma 3, which states that a reference w corresponding to a location y has the read-only version of the type of y .

► **Lemma 3 (RefT).** *If $F; \rho \vdash y : T_1$, and $F; \rho \vdash T_1 \text{ ro } T_2$, and $F \sim \rho$, and $w \rightarrow y \in \rho$, then $F; \rho \vdash w : T_2$.*

Showing the same for the mutability part of the type is easy, because it is formed in the same way as in the TT-Read term typing rule.

The rule also changes the runtime environment ρ . Correspondence of ρ with the typing context is ensured because w is given the same type as y except for mutability.

In the TR-LetNew rule, it must be shown that heap correspondence is preserved. That is, the object on the heap has the type given to the new location y added to F . First, we show that definitions keep their type if references are replaced by locations, by Lemma 4. Then, we use a variant of a substitution Lemma 5 to show that if the definitions d of the object had type T_1 given by the DT-* definition typing rules, then under substitution of the location y for the self variable s , the definition will get the type by the HT-* typing rules. Preservation of ρ correspondence is ensured by giving w_1 the same type as y_1 .

► **Lemma 4 (DeD).** *If $F, s : T_2; \rho \vdash d : T_1$, and $F \sim \rho$, then $F, s : T_2; \rho \vdash [\rho]d : T_1$.*

► **Lemma 5 (SubD).** *If $\Gamma, s : T_3; \rho \vdash d : T_1$, and $s \notin \Gamma$ and $\Gamma \text{ vis } y$ and $\Gamma; \rho \vdash y : [y/s]T_3$, then $\Gamma, y/s : T_3; \rho \vdash [y/s]d : [y/s]T_1$.*

Finally, weakening lemmata state that adding variables to the typing context preserves typing derivations.

With progress and preservation lemmata proven for individual cases, we can state a common progress and preservation Theorem 6.

► **Theorem 6 (TPP).** *If $F_1 \vdash \langle t_1; \sigma_1; \rho_1; \Sigma_1 \rangle : T$, then either $\langle t_1; \sigma_1; \rho_1; \Sigma_1 \rangle = \langle \nu w_1; \cdot; \rho_1; \Sigma_1 \rangle$, or there exist $t_2, \sigma_2, \Sigma_2, \rho_2, F_2$, such that $F_1 \vdash \langle t_1; \sigma_1; \rho_1; \Sigma_1 \rangle : T \mapsto F_1, F_2 \vdash \langle t_2; \sigma_2; \rho_2; \Sigma_2 \rangle$, and $F_1, F_2 \vdash \langle t_2; \sigma_2; \rho_2; \Sigma_2 \rangle : T$.*

By induction on the number of steps, type soundness of typed reduction follows – Theorem 7. Because typed reduction affects typed configurations in the same way as untyped reduction, we can easily show the final type soundness for untyped reduction Theorem 8.

► **Theorem 7 (TyS).** *If $\vdash t_0 : T$, then either $\exists w, j, \Sigma, \rho, F: \vdash \langle t_0; \cdot; \cdot; \cdot \rangle : T \mapsto^j F \vdash \langle \nu w; \cdot; \rho; \Sigma \rangle$ or $\forall j: \exists t_j, \sigma_j, \Sigma_j, \rho_j, F_j: \vdash \langle t_0; \cdot; \cdot; \cdot \rangle : T \mapsto^j F_j \vdash \langle t_j; \sigma_j; \rho_j; \Sigma_j \rangle$.*

► **Theorem 8 (S).** *If $\vdash t_0 : T$, then either $\exists w, j, \Sigma, \rho: \langle t_0; \cdot; \cdot; \cdot \rangle \mapsto^j \langle \nu w; \cdot; \rho; \Sigma \rangle$ or $\forall j: \exists t_j, \sigma_j, \Sigma_j, \rho_j: \langle t_0; \cdot; \cdot; \cdot \rangle \mapsto^j \langle t_j; \sigma_j; \rho_j; \Sigma_j \rangle$.*

5.2.2 Proof of the Immutability Guarantee

A new essential property of the type system is the immutability guarantee, expressed by Theorem 9. It says that if an object exists in a typed configuration c_1 , then either it is mutably reachable by **mreach** (and therefore can change), or it stays the same after any number of execution steps (never changes).

► **Theorem 9 (IG).** *If $y \rightarrow d \in \Sigma_1$, and $F_1 \vdash \langle t_1; \sigma_1; \rho_1; \Sigma_1 \rangle : T$, and $\langle t_1; \sigma_1; \rho_1; \Sigma_1 \rangle \mapsto^k \langle t_2; \sigma_2; \rho_2; \Sigma_2 \rangle$, then either $y \rightarrow d \in \Sigma_2$ or $F_1 \vdash \langle t_1; \sigma_1; \rho_1; \Sigma_1 \rangle \mathbf{mreach} y$.*

For the proof, we again make use of the typed reduction rules from Figure 13. We show two properties of **mreach**: First, Lemma 10 states that if an object is mutated in a reduction step, then it was mutably reachable before the step. Second, Theorem 11 states that **mreach** is preserved by typed reduction, that is, an existing object that is not **mreach** will never become **mreach** in the future. This has to be shown for each of the reduction rules.

► **Lemma 10 (MMR).** *If $F_1 \vdash \langle t_1; \sigma_1; \rho_1; \Sigma_1 \rangle : T \mapsto F_2 \vdash \langle t_2; \sigma_2; \rho_2; \Sigma_2 \rangle$, and $y \rightarrow d \in \Sigma_1$, then either $y \rightarrow d \in \Sigma_2$ or $F_1 \vdash \langle t_1; \sigma_1; \rho_1; \Sigma_1 \rangle \mathbf{mreach} y$.*

► **Theorem 11 (MP).** *If $F_1 \vdash \langle t_1; \sigma_1; \rho_1; \Sigma_1 \rangle : T \mapsto F_2 \vdash \langle t_2; \sigma_2; \rho_2; \Sigma_2 \rangle$, $y \rightarrow d \in \Sigma_1$, and $F_2 \vdash \langle t_2; \sigma_2; \rho_2; \Sigma_2 \rangle \mathbf{mreach} y$, then $F_1 \vdash \langle t_1; \sigma_1; \rho_1; \Sigma_1 \rangle \mathbf{mreach} y$.*

The proof of Lemma 10 is straightforward from the reduction and typing rules, because only the TR-Write rule modifies existing objects on the heap and the typing rule for write terms requires the object reference to have a read-write type.

The rest of this section is about proving Theorem 11. For each of the 6 reduction rules, we look at the changes in the configuration and typing context that affect the **mreach** relation.

In the TR-LetPush and TR-LetLoc rules, the only difference in the configuration relevant to **mreach** is that t-free object references are moved between the focus and the stack. No new references are introduced, and the heap, environment and context do not change. These cases are handled by Lemma 12, which states that under these conditions, no object becomes mutably reachable.

► **Lemma 12 (MPres).** *If $F \vdash \langle t_2; \sigma_2; \rho; \Sigma \rangle \mathbf{mreach} y$, and $\forall x: (t_2 \mathbf{tfree} x \vee \sigma_2 \mathbf{tfree} x) \Rightarrow (t_1 \mathbf{tfree} x \vee \sigma_1 \mathbf{tfree} x)$, then $F \vdash \langle t_1; \sigma_1; \rho; \Sigma \rangle \mathbf{mreach} y$.*

18:22 Reference Mutability for DOT

For TR-Apply, Lemma 12 also applies, because the HT-Met rule ensures that variables other than the receiver and the argument are not visible, and therefore cannot be t-free in the body of the method.

In TR-Write, a location may be stored on the heap as a new value of a read-write field. The typing of write terms ensures that if the field is read-write, then the reference w_3 that provided the value was read-write, so the location y_3 was mutably reachable from the focus of execution.

A TR-LetNew step creates a new mutably reachable object. New objects are not covered by the immutability guarantee, but the new object may contain read-write fields referring to existing objects. Similarly to the Write case, the typing of field definitions in the object literal ensures that if the fields have read-write type, then the references in the literal must have been read-write.

The TR-LetNew rule also adds a new location and reference to the typing context. The preservation of **mreach** depends on an essential property of how mutability is defined: *existing* read-only references and fields cannot be made read-write by creating new objects and references. Lemma 13 states that if a variable v_1 is read-write in an inert context F with a new location y_2 added, then it was already read-write in the context F without y_2 . (In other words, it keeps its mutability if the last location y_2 is removed from the context.) We state Lemma 14 for mutability of fields and similar Lemmata 15 and 16 for adding a new reference w_2 to F and ρ .

► **Lemma 13** (StnMLoc). *If $v_1 \neq y_2$, and $F_2 = F_1, y_2 : T$, and $F_2; \rho \vdash v_1 : \{M : \perp.. \perp\}$, then $F_1; \rho \vdash v_1 : \{M : \perp.. \perp\}$.*

► **Lemma 14** (StnMFLoc). *If $y_1 \neq y_2$, and $F_2 = F_1, y_2 : T$, and $F_2; \rho \vdash y_1 : \{a : \perp.. \{M : \perp.. \perp\}\}$, then $F_1; \rho \vdash y_1 : \{a : \perp.. \{M : \perp.. \perp\}\}$.*

► **Lemma 15** (StnMRef). *If $v_1 \neq w_2$, and $F_2 = F_1, w_2 : T$, and $\rho_2 = \rho_1, w_2 \rightarrow y_2$, and $F_2 \sim \rho_2$, and $F_2; \rho_2 \vdash v_1 : \{M : \perp.. \perp\}$, then $F_1; \rho_1 \vdash v_1 : \{M : \perp.. \perp\}$.*

► **Lemma 16** (StnMRef). *If $y_1 \neq w_2$, and $F_2 = F_1, w_2 : T$, and $\rho_2 = \rho_1, w_2 \rightarrow y_2$, and $F_2 \sim \rho_2$, and $F_2; \rho_2 \vdash y_1 : \{a : \perp.. \{M : \perp.. \perp\}\}$, then $F_1; \rho_1 \vdash y_1 : \{a : \perp.. \{M : \perp.. \perp\}\}$.*

The case of adding a new *reference* to an existing location has a quite straightforward proof: in the typing derivation that gives a read-write type in the new context, we can replace the new reference by the corresponding location.

The case of adding a new *location* y_2 is more complicated, because the location has a type that may not be present anywhere else in the context. We can use the infrastructure of invertible typing to show that the mutability of a reference w_1 , location y_1 , or its field a can be derived from the type specified for w_1 or y_1 in F by tight subtyping. That is sufficient to prove Lemma 13, but in Lemma 14, we still need to show that the upper bound given to $y_1.a$ in the typing context is a tight subtype of $\{M : \perp.. \perp\}$. It is not possible to show that for subtyping of arbitrary types, even if both types do not reference the variable y_2 . In particular, this is caused by subtyping of method types, where subtyping of the result type may be in a typing context that is not inert. Fortunately, we need this property only for the special case when the right-hand side of the subtyping is the simple type $\{M : \perp.. \perp\}$, as stated by Lemma 17. The premise T_2 **nosel** y_2 means that T_2 does not contain type selections involving y_2 . The # sign indicates the use of tight subtyping.

► **Lemma 17** (StnSub). *If $F_2 = F_1, y_2 : T_1$, and $F_2; \rho \vdash_{\#} T_2 <: \{M : \perp.. \perp\}$, and T_2 **nosel** y_2 , then $F_1; \rho \vdash T_2 <: \{M : \perp.. \perp\}$.*

Proof sketch of Lemma 17. We want to show that the subtyping derivation never needs to use y_2 , that is, that we can derive the same subtyping without invoking the $\text{ST}_{\#}\text{-SelL}$ or $\text{ST}_{\#}\text{-SelU}$ rules on selections from y_2 . There are 3 ways in which the original derivation may involve y_2 : using the $\text{ST}_{\#}\text{-SelL}$ or $\text{ST}_{\#}\text{-SelU}$ selection rules, using subtyping of method types, or using rules such as ST-Top or ST-And1 , where one of the types may be chosen freely. The proof proceeds in 3 steps, where in each step, we eliminate one of these issues by simplifying the types on both sides of the subtyping, and showing that the simplified types are related by a restricted version of subtyping.

For the first step, we define a reduction relation \mapsto^s . Because in an inert context, bounds are either tight or have a lower bound of \perp , using the $\text{ST}_{\#}\text{-Sel}^*$ rules does not add anything that could not be derived by other subtyping rules. Therefore, we can get rid of unnecessary uses of $\text{ST}_{\#}\text{-Sel}^*$ in the subtyping derivation by replacing type selections by their bounds. The reduction has two variants, \mapsto_{\oplus}^s and \mapsto_{\ominus}^s , that replace a type selection by its upper (respectively lower) bound. The restricted version of subtyping allows using the selection rules only in the direction from the selection to the bound, not vice versa.

In the second step, we show that subtyping of method types is not needed using a reduction relation \mapsto^m that replaces all method types by \top or \perp . The restricted version of subtyping does not have the ST-Met rule.

In the last step, we show that y_2 never has to appear in the types involved in the derivation of the subtyping taken from left to right. The reduction \mapsto^e replaces the remaining selections on y_2 by \top or \perp .

In each of these steps, the type on the left-hand side, starting with T_2 , is simplified to a supertype, and the $\{M : \perp.. \perp\}$ on the right hand side is not affected by the simplification. Therefore after all the steps, we get $F_1; \rho \vdash T_2 <: \{M : \perp.. \perp\}$. \blacktriangleleft

Finally, in the TR-Read rule, a new reference is added to the context. The effect of adding the reference to the typing context is handled by Lemmata 15 and 16. A final piece in the proof of **mreach** preservation is mutability of the new reference w_2 created in a TR-Read step. It is created for a location that was stored in a field and is put into the focus of execution. We must ensure that the reference is read-write only if the field was read-write. Because the mutability of w_2 is a union of the field mutability and the mutability of the source reference w_1 , we first show that if w_2 is read-write, then both the field and the source are read-write. For the field, we use Lemma 18 to show that the field has type $\{a : \perp.. \{M : \perp.. T_7\}\}$, and then by Lemma 16, it must have had the same type before.

► **Lemma 18 (MUSub).** *If $\Gamma; \rho \vdash T_1 \text{ mu}(r) T_2$, then $\Gamma; \rho \vdash \top <: T_2$ or $\Gamma; \rho \vdash T_1 <: \{M(r) : \perp.. T_2\}$.*

For the object, we need Lemma 19 to show that if the upper bound of $w_1.M$ in the new context is \perp , then w_1 was a read-write reference in the old context F .

► **Lemma 19 (WMu).** *If $w_1 \neq w_2$, and $F, w_2 : T_2; \rho_2 \vdash_{\#} w_1.M(r) <: \perp$, and $\rho_2 = \rho_1, w_2 \rightarrow y_2$, and $F, w_2 : T_2 \sim \rho_2$, then $F; \rho_1 \vdash w_1 : \{M(r_2) : \perp.. \perp\}$.*

We use the \mapsto_{\oplus}^s relation from the proof of Lemma 17 above to show this. It simplifies $w_1.M(r)$ to its bound: $F, w_2 : T_2 \vdash w_1.M(r) \mapsto_{\oplus}^s T_1$. Then, by further simplifying both sides of the subtyping, we find a type which is a supertype of T_1 and subtype of \perp in F .

6 Related Work

6.1 Reference Mutability

Scala is influenced by Java, which has seen several extensions for reference mutability.

Javari [19] extends the Java syntax with reference mutability qualifiers. An unqualified reference type T is by default a read-write reference, while `readonly T` is a read-only reference. Javari comes with both a formal system based on Featherweight Java [11], and an implementation in the Checker Framework [13], using type annotations. The type system provides a transitive immutability guarantee, but allows opting out of that by declaring fields as always assignable, even through read-only references, or as always read-write, meaning viewpoint adaptation does not apply to them. (Non-transitive immutability could be achieved in the framework of our type system by removing the viewpoint adaptation in the typing rule for read terms and changing the definition of mutable reachability.) Qualifiers can be applied to any type in the program. Qualifier polymorphism is limited to the `romaybe` qualifier, which acts as a variable qualifier which can be instantiated at use locations by `mutable` or `readonly` – all `romaybe` qualifiers in a method declaration by the same qualifier. This allows Javari to express the first example T_{o1} from Section 4.5:

```
class C {
  T a;
  void m_set(T z) {a = z;}
  romaybe T m_get() romaybe {return a;}
}
```

The field is by default this-mutable and the `m_set` method is by default mutable with a mutable parameter. Javari allows mutability to be part of a type argument, so we could make the field `a` mutability-polymorphic like in the second example T_{o2} , but we would not be able to express the full example because Javari has no way to combine the mutability of the field with the mutability of the receiver of `m_get`.

The type system of ReIm [10] is simpler than that of Javari to enable fast and scalable inference of qualifiers. It has only 3 qualifiers, `mutable`, `readonly` and `polyread`. The `polyread` qualifier expresses simple qualifier polymorphism and viewpoint adaptation, similar to `romaybe` in Javari. Fields are either `readonly` or `polyread`; always-read-write fields are not supported. Usage of qualifiers is limited – they can be applied to any type, but not to type arguments of generic types. The first example T_{o1} can be expressed in ReIm as follows:

```
class C {
  polyread T a;
  void m_set(mutable C this, mutable T z) {a = z;}
  polyread T m_get(polyread C this) {return a;}
}
```

The second example T_{o2} cannot be expressed due to the lack of qualifiers on type arguments.

Immutable Generic Java (IGJ) [20] encodes mutability qualifiers in Java generics. It defines the first parameter of a class or interface to specify its mutability: the type `T<Mutable>` is read-write and the type `T<ReadOnly>` is read-only. This approach agrees with our desire to use features of the underlying type system to specify reference mutability. IGJ does not have viewpoint adaptation. Transitivity has to be opted into by declaring fields with a “this-mutable” type, using the mutability parameter of the containing class. As in Javari, fields

may be declared always assignable. IGJ also supports object immutability by distinguishing `ReadOnly` references and `Immutable` references. The latter guarantee that the object will not be modified through any reference. Our first example T_{o1} can be expressed in IGJ as follows by explicitly specifying viewpoint adaptation in the return type of the getter method:

```
class C<I extends ReadOnly> {
  T<Mutable> a;
  @Mutable void m_set(T<Mutable> z) {a = z;}
  @ReadOnly T<I> m_get() {return a;}
}
```

The second example T_{o2} cannot be fully expressed for the same reason as in Javari.

Glacier [5] has a system based on class immutability. It has only two qualifiers that apply to classes. An `@Immutable` class must only have immutable subclasses and all fields must have immutable types. All other classes are `@MaybeMutable`. Class types other than the top class `Object` cannot be qualified when used and always have the mutability declared by the class.

The type systems above were implemented in the Checker Framework. This framework expresses type qualifiers using Java annotations, so that the Java syntax does not have to be modified. Qualifiers that apply to the receiver of a method are written by annotating the explicit `this` parameter. We achieve the same result in our approach using the explicit receiver parameter to a method. Explicit `this` parameters are not supported in Scala.

The type systems above share the limitations of Java generics and of Java; in particular, they do not support type intersections and unions.

The reference mutability system for the C# language [7] is the most flexible. As in the systems above, a type is composed of a qualifier and a normal type. In this type system, a generic class can be parameterized by both normal types and type qualifiers, but separately, by declaring a second qualifier parameter list after the type parameter list. Therefore, a class may have any number of qualifier parameters, which can be used to individually specify mutability of fields, method parameters and result types, or be passed as qualifier arguments to the types used at those places. Qualifiers can be combined by the special type operator \rightsquigarrow , which viewpoint adapts the second qualifier by the first one. This makes it possible to express a class similar to our second example T_{o2} from Section 4.5 as follows:

```
class C<PT> {
  PT T a;
  void Ms(PT T z) writable {a = z;}
  PC $\rightsquigarrow$ PT T Mg<_><PC>() PC {return a;}
}
```

Other supported features include object immutability and uniqueness in a multi-threaded context for safe parallelism.

6.2 DOT

In traditional DOT calculi, such as WadlerFest DOT, objects are immutable and their fields cannot be changed. A heap is not needed because object literals can be used directly as values in terms.

Mutable WadlerFest DOT [15] introduced mutability by means of mutable cells, which are allocated to hold a single variable and can be reassigned to other variables. In this scheme, an equivalent of a mutable field can be achieved by having a field which contains a mutable cell. Although the field itself cannot be reassigned, writing a value to this cell and reading the value of the cell behaves as writing and reading a value from a mutable field.

A move towards a more direct definition of mutable fields has been made in kDOT [12], where all objects are stored on the heap and referred to by object locations, and field assignments through such locations directly modify the object on the heap. This approach is closer to how object mutation works in Scala. With slight modifications, we used it as the baseline for our type system.

6.3 Programming Languages with Reference Mutability

Some programming languages have reference mutability as a part of their type system.

The const-qualified pointers and methods in the C++ programming language disallow mutation of the object pointed to [18]. However, `const` is not transitive, so it does not correspond to our definition of immutability. There is no concept of a viewpoint-adapted field. Qualifier polymorphism is not supported, but can be achieved using templates, or by directly duplicating the definitions in source code.

The D language has transitive `const` and `immutable` type qualifiers, which express reference and object immutability [1]. Like in the Java-based systems, D does not have intersection, union and dependent types, and objects have class types. Qualifier polymorphism is limited to templates combined with D's advanced support for support of metaprogramming and compile-time evaluation.

The Pony programming language defines reference capabilities, which qualify the type of every reference [4, 17]. The system is defined in the context of multiple simultaneously running actors, and by allowing only one actor to have a read-write reference or multiple actors to have read-only references to an object, it ensures that no race conditions can occur when modifying an object. The qualifiers not only specify whether the reference can be used to mutate an object, but also limit which other references to the same object may exist within the same or a different actor. Qualifiers corresponding to read-write and read-only references as used in this paper would be `ref` and `box`. Pony also has viewpoint adaptation applied to types of fields, and it can be also used from source code by writing arrow types, which allow viewpoint adapting a type by a type parameter, `this`, or `box`.

In Rust, mutability is tied to ownership. There can only be one mutable reference to an object. Read-only references are transitive. Although a reference can be qualified by lifetime parameters, its mutability is fixed, so there is no mutability polymorphism.

7 Conclusion

We have extended the DOT calculus with support for reference mutability. Our calculus, roDOT, supports a transitive immutability guarantee using viewpoint adaptation. Unlike most existing reference mutability systems that have been proposed for Java, which are expressed as a separate type system in parallel to the Java type system, our system is encoded using existing features of the DOT type system. Specifically, the mutability of an object is encoded using a type member, which makes it possible to refer to it in other types using a path-dependent type. An important and necessary enhancement to DOT is a separate notion of a reference as opposed to just a heap location, which makes it possible to distinguish mutable and read-only references to the same location. We have proven type soundness of roDOT, as well as the immutability guarantee that it provides. The calculus can serve as a formal foundation for future work on a principled implementation of reference mutability in a Scala compiler.

References

- 1 Andrei Alexandrescu. *The D programming language*. Addison-Wesley, Upper Saddle River, N.J., 2009.
- 2 Nada Amin, Samuel Grütter, Martin Odersky, Tiark Rompf, and Sandro Stucki. The essence of dependent object types. In Sam Lindley, Conor McBride, Philip W. Trinder, and Donald Sannella, editors, *A List of Successes That Can Change the World - Essays Dedicated to Philip Wadler on the Occasion of His 60th Birthday*, volume 9600 of *Lecture Notes in Computer Science*, pages 249–272. Springer, 2016. doi:10.1007/978-3-319-30936-1_14.
- 3 Nada Amin, Tiark Rompf, and Martin Odersky. Foundations of path-dependent types. 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 233–249. ACM, 2014. doi:10.1145/2660193.2660216.
- 4 Sylvan Clebsch, Sophia Drossopoulou, Sebastian Blessing, and Andy McNeil. Deny capabilities for safe, fast actors. In Elisa Gonzalez Boix, Philipp Haller, Alessandro Ricci, and Carlos Varela, editors, *Proceedings of the 5th International Workshop on Programming Based on Actors, Agents, and Decentralized Control, AGERE! 2015, Pittsburgh, PA, USA, October 26, 2015*, pages 1–12. ACM, 2015. doi:10.1145/2824815.2824816.
- 5 Michael J. Coblentz, Whitney Nelson, Jonathan Aldrich, Brad A. Myers, and Joshua Sunshine. Glacier: transitive class immutability for java. In Sebastián Uchitel, Alessandro Orso, and Martin P. Robillard, editors, *Proceedings of the 39th International Conference on Software Engineering, ICSE 2017, Buenos Aires, Argentina, May 20-28, 2017*, pages 496–506. IEEE / ACM, 2017. doi:10.1109/ICSE.2017.52.
- 6 Vlastimil Dort and Ondřej Lhoták. Reference mutability for DOT - roDOT definitions and proofs. Technical Report D3S-TR-2020-01, Dep. of Distributed and Dependable Systems, Charles University, 2020.
- 7 Colin S. Gordon, Matthew J. Parkinson, Jared Parsons, Aleks Bromfield, and Joe Duffy. Uniqueness and reference immutability for safe parallelism. In *Proceedings of the 27th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2012, part of SPLASH 2012, Tucson, AZ, USA, October 21-25, 2012*, pages 21–40, 2012. doi:10.1145/2384616.2384619.
- 8 Philipp Haller and Ludvig Axelsson. Quantifying and explaining immutability in scala. In *Proceedings Tenth Workshop on Programming Language Approaches to Concurrency- and Communication-cEntric Software, PLACES@ETAPS 2017, Uppsala, Sweden, 29th April 2017*, pages 21–27, 2017. doi:10.4204/EPTCS.246.5.
- 9 Jason Z. S. Hu and Ondřej Lhoták. Undecidability of D<: and its decidable fragments. *PACMPL*, 4(POPL):9:1–9:30, 2020. doi:10.1145/3371077.
- 10 Wei Huang, Ana Milanova, Werner Dietl, and Michael D. Ernst. ReIm & ReImInfer: checking and inference of reference immutability and method purity. In *Proceedings of the 27th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2012, part of SPLASH 2012, Tucson, AZ, USA, October 21-25, 2012*, pages 879–896, 2012. doi:10.1145/2384616.2384680.
- 11 Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler. Featherweight Java: a minimal core calculus for Java and GJ. *ACM Trans. Program. Lang. Syst.*, 23(3):396–450, 2001. doi:10.1145/503502.503505.
- 12 Ifaz Kabir and Ondřej Lhoták. κDOT: scaling DOT with mutation and constructors. In *Proceedings of the 9th ACM SIGPLAN International Symposium on Scala, SCALA@ICFP 2018, St. Louis, MO, USA, September 28, 2018*, pages 40–50, 2018. doi:10.1145/3241653.3241659.
- 13 Matthew M. Papi, Mahmood Ali, Telmo Luis Correa Jr., Jeff H. Perkins, and Michael D. Ernst. Practical pluggable types for Java. In Barbara G. Ryder and Andreas Zeller, editors, *Proceedings of the ACM/SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2008, Seattle, WA, USA, July 20-24, 2008*, pages 201–212. ACM, 2008. doi:10.1145/1390630.1390656.

- 14 Marianna Rapoport, Ifaz Kabir, Paul He, and Ondřej Lhoták. A simple soundness proof for dependent object types. *PACMPL*, 1(OOPSLA):46:1–46:27, 2017. doi:10.1145/3133870.
- 15 Marianna Rapoport and Ondřej Lhoták. Mutable WadlerFest DOT. In *Proceedings of the 19th Workshop on Formal Techniques for Java-like Programs, Barcelona , Spain, June 20, 2017*, pages 7:1–7:6, 2017. doi:10.1145/3103111.3104036.
- 16 Tiark Rumpf 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, part of SPLASH 2016, Amsterdam, The Netherlands, October 30 - November 4, 2016*, pages 624–641. ACM, 2016. doi:10.1145/2983990.2984008.
- 17 George Steed. A principled design of capabilities in Pony. https://www.ponylang.io/media/papers/a_principled_design_of_capabilities_in_pony.pdf.
- 18 Bjarne Stroustrup. *The C++ programming language*. Addison-Wesley, Upper Saddle River, NJ, 2013.
- 19 Matthew S. Tschantz and Michael D. Ernst. Javari: adding reference immutability to Java. In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2005, October 16-20, 2005, San Diego, CA, USA*, pages 211–230, 2005. doi:10.1145/1094811.1094828.
- 20 Yoav Zibin, Alex Potanin, Mahmood Ali, Shay Artzi, Adam Kiezun, and Michael D. Ernst. Object and reference immutability using Java generics. In *Proceedings of the 6th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2007, Dubrovnik, Croatia, September 3-7, 2007*, pages 75–84, 2007. doi:10.1145/1287624.1287637.


Tackling the Awkward Squad for Reactive Programming: The Actor-Reactor Model

Sam Van den Vonder 

Software Languages Lab, Vrije Universiteit Brussel, Belgium

Thierry Renaux 

Software Languages Lab, Vrije Universiteit Brussel, Belgium

Bjarno Oeyen 

Software Languages Lab, Vrije Universiteit Brussel, Belgium

Joeri De Koster 

Software Languages Lab, Vrije Universiteit Brussel, Belgium

Wolfgang De Meuter 

Software Languages Lab, Vrije Universiteit Brussel, Belgium

Abstract

Reactive programming is a programming paradigm whereby programs are internally represented by a dependency graph, which is used to automatically (re)compute parts of a program whenever its input changes. In practice reactive programming can only be used for some parts of an application: a reactive program is usually embedded in an application that is still written in ordinary imperative languages such as JavaScript or Scala. In this paper we investigate this embedding and we distill “the awkward squad for reactive programming” as 3 concerns that are essential for real-world software development, but that do not fit within reactive programming. They are related to long lasting computations, side-effects, and the coordination between imperative and reactive code. To solve these issues we design a new programming model called the Actor-Reactor Model in which programs are split up in a number of actors and reactors. Actors and reactors enforce a strict separation of imperative and reactive code, and they can be composed via a number of composition operators that make use of data streams. We demonstrate the model via our own implementation in a language called Stella.

2012 ACM Subject Classification Software and its engineering → Data flow languages; Software and its engineering → Multiparadigm languages

Keywords and phrases functional reactive programming, reactive programming, reactive streams, actors, reactors

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2020.19

Supplementary Material ECOOP 2020 Artifact Evaluation approved artifact available at <https://doi.org/10.4230/DARTS.6.2.7> and <https://doi.org/10.5281/zenodo.3862954>.

Funding *Sam Van den Vonder*: Research Foundation – Flanders (FWO) grant No. 1S95318N
Thierry Renaux: Flanders Innovation & Entrepreneurship (VLAIO) “Cybersecurity Initiative Flanders” program
Bjarno Oeyen: Research Foundation – Flanders (FWO) grant No. 1S93820N

1 Introduction

Reactive programming is a programming paradigm that revolves around automatically changing the state of a program based on perpetually incoming values. Historically, it was conceived as a solution to the problems of *inversion of control* and *callback hell* which occur when programming event-driven programs [3]. Reactive programming languages such as FrTime [12], Flapjax [34], Elm [15] and REScala [46] propose increasingly rich abstraction



© Sam Van den Vonder, Thierry Renaux, Bjarno Oeyen, Joeri De Koster, and Wolfgang De Meuter;
licensed under Creative Commons License CC-BY

34th European Conference on Object-Oriented Programming (ECOOP 2020).

Editors: Robert Hirschfeld and Tobias Pape; Article No. 19; pp. 19:1–19:29

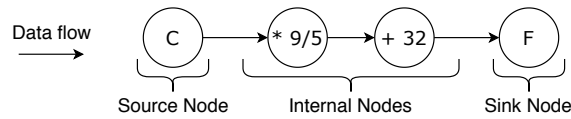
Leibniz International Proceedings in Informatics



Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany



19:2 Tackling the Awkward Squad for Reactive Programming



■ **Figure 1** Compiled structure of a reactive program to a DAG. Data flow is usually depicted from top to bottom, but depicted here from left to right.

mechanisms to represent and compose so-called “time-varying values” or *signals*, which are values that can change over time. The “Hello World!” of reactive programming converts measurements of a Celsius thermometer to Fahrenheit. Given a signal C that represents a changing temperature in Celsius, the expression $F = (C * 9/5) + 32$ declares a new signal F that represents the temperature in Fahrenheit. Changes to the value of C automatically give rise to a recomputation of F . This is typically realised by compiling the reactive program into a directed acyclic graph (DAG), as exemplified in Figure 1.

There are many incarnations of reactive languages and libraries built with various mainstream languages. In academia, reactive languages include FrTime [12] (Racket), Scala.React [33] (Scala), REScala [46] (Scala) and Flapjax [34] (JavaScript). In industry, large companies such as Facebook develop and maintain frameworks such as ReactJS [28] and React Native [21] for reactive Graphical User Interfaces (GUIs), ReactiveX is a specification for reactive streams implemented in over 18 languages [40] some of which are developed or maintained by companies such as Microsoft and Netflix, and an implementation of the “Reactive Streams” specification has been included in Java since version 9 [29, 13]. Results from an empirical study on program comprehension suggest favourable results for reactive programming when compared to the Observer pattern in object-oriented programming [47].

In the rest of the paper we will abstract over the details of particular reactive programming languages and talk about them in terms of their role in the canonical DAG model for reactive programming. *Source nodes* correspond to the “input” signals of the reactive program. Their values are typically provided by code that is external to the reactive program. *Internal nodes* are composed signals that constitute the reactive program. *Sink nodes* correspond to the “output” signals of the reactive program that constitute the program output.

Reactive programming languages and frameworks focus on the design and concepts of the *internal part* of a reactive program. Explained in terms of the DAG, they focus on language features and abstractions whereby programmers can express DAGs as easily and as declaratively as possible. Reactive programs also have 2 other parts: an *input part* provides input to the reactive program by modifying its source nodes, and another (possibly different) *output part* processes the output of the reactive program. For example, in the temperature converter of Figure 1, the source C may be connected to an input field in the GUI, and the result F may be displayed in the same GUI. In another application, C may be connected to a distributed web-based data stream whereby the input of the reactive program is produced by an entirely different machine (e.g. a weather station).

One cannot help but observe that reactive programming is only used to implement the internal part of a reactive program, and that it is usually embedded in an application that is still written in ordinary imperative languages such as JavaScript or Scala. The main problem tackled in this paper is that the parts of existing reactive programming languages and frameworks that are responsible for input and output are ill-defined: They use ad-hoc mechanisms that can violate the invariants of reactive programs. This is especially problematic for long lasting computations that block the reactive program [7], and computations with

■ **Listing 1** REScala reactive program that matches a regular expression with an input string.

```
1 val userInputSignal = Var("") // initial signal value is ""
2 val matches = Signal { "(A+)*B".r.findFirstMatchIn(userInputSignal()) }
```

side-effects (e.g. modifying or rendering a GUI) [19, 33]. Mechanisms to embed reactive code between the imperative input and output parts of applications are poorly investigated in literature. To this end, we have 2 main contributions.

1. In analogy with the “awkward squad” for functional programming which are a set of application concerns that are essential for real-world software development, but that do not fit within the purely functional programming paradigm [37], in Section 2 we identify the “*awkward squad for reactive programming*”. They are (1) long lasting computations, (2) embedding imperative code in reactive code, and (3) embedding reactive code in imperative code.
2. Just like functional programming solves the problem by evacuating the awkward squad to a different location (i.e. monadic) of the program, we propose a programming model that solves these issues. First, in Section 3 we define a class-based object-oriented data model that will guarantee that reactive programs cannot execute imperative code and long lasting computations. Second, in Section 4 we introduce the *Actor-Reactor Model* whereby the imperative input & output parts of reactive programs must be modelled as *actors*, and the internal parts of reactive programs are modelled as *reactors*. Together, actors and reactors enforce that imperative and reactive code remains separated, but can still co-exist within the same application. To clearly demonstrate the concepts of this model we have designed and implemented an experimental language called Stella.

2 Identifying the Awkward Squad for Reactive Programming

In this section we analyse the problems that may occur when embedding a reactive programming model in an imperative programming language, or when allowing the internal nodes of a reactive program to be programmed with the full power of a Turing-complete language.

2.1 Long Lasting Computations

Responsiveness is 1 of the 4 key properties of reactive systems outlined in the so-called Reactive Manifesto [7], [31, Chapter 1]: “*Responsive systems focus on providing rapid and consistent response times, establishing reliable upper bounds so they deliver a consistent quality of service.*” [7] However, reactive languages and frameworks often impose little to no restrictions on the types of expressions that can be used within a reactive program. It is often easy to write code that accidentally makes the program no longer reactive. For example, consider the reactive program in Listing 1 (written in REScala) that checks whether user-provided input strings match a regular expression. Line 1 defines a new source node called `userInputSignal`, and Line 2 defines an internal node called `matches` that is derived from the source node. Whenever the value of `userInputSignal` changes, the value of `matches` reflects whether the new string matches the regular expression $(A+)*B$ (e.g. `AB` and `AAAB`, but not `AAA`). This program has a worst-case complexity of $\mathcal{O}(2^n)$ with n the size of the input string. Matching the string `AAAAA` fails after 112 steps, and matching 50 `A`'s fails only after ~ 3 quadrillion steps [44]. This program clearly cannot be called reactive in the aforementioned sense. While this specific example may be a contrived case of catastrophic backtracking in

19:4 Tackling the Awkward Squad for Reactive Programming

regular expressions, a developer can easily and accidentally introduce computations into reactive programs that (occasionally) have unintended consequences for their execution time. We call this the **Reactive Thread Hijacking Problem**, because long lasting computations can “hijack” the thread of execution of a reactive program, thereby stopping the reactive program from being able to react to new input.

The language design problem that needs to be solved is how to ensure that a “reliable upper bound” can be imposed on long lasting computations within reactive programs.

The Reactive Manifesto is intentionally vague about how to achieve this. We identified 3 levels of reactivity that provide different program termination guarantees.

2.1.1 Weak Reactivity

The set of programs that we call *weakly reactive* are those for which there are no guarantees with respect to how long they will take to execute. This is usually because reactive programs can be programmed with the full power of a Turing-complete language. Programs written in reactive languages such as FrTime [12], Flapjax [34], Elm [15], REScala [46] and AmbientTalk/R [9], or frameworks such as ReactJS [28], ReactiveX [40] and Akka Streams [45, Chapter 13] all fall into this category.

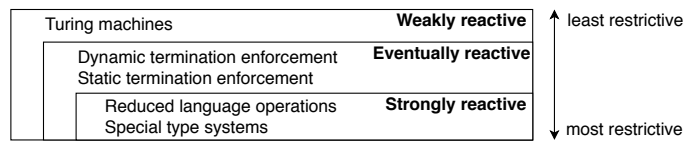
2.1.2 Eventual Reactivity

The set of programs that we call *eventually reactive* are those for which it can be proven that the program will eventually terminate. There are both static and dynamic approaches to enforce program termination, each with varying levels of restrictions that they impose on the underlying program. Static enforcement includes work such as total functional programming [51], primitive recursion [8, Chapter 3], and model checkers such as TERMINATOR [11]. Techniques that dynamically enforce program termination can make use of both static information and run-time values, such as size-change termination [35].

2.1.3 Strong Reactivity

The set of programs that we call *strongly reactive* are those for which the execution time of the reactive program does not depend on the size of its input. In other words, the asymptotic worst-case complexity of the reactive program is guaranteed to be in $\mathcal{O}(1)$. Reactive programs that are strongly reactive will never be unexpectedly slow because of certain types of input. We consider this to be the strongest form of reactivity, but the trade-off is that the types of programs that can be written is severely restricted.

Synchronous programming languages such as Esterel [5], Lustre [25], C  u [48] and Signal [23] apply reactive programming to real-time systems [4]. They rely on the assumption of the *synchronous hypothesis*, where the output of the program is conceptually synchronous with its input and “instantaneous” [17]. In contrast with strong reactivity this is not a constraint on program complexity or execution time, but on the correctness of event processing whereby 1 event must be completely processed by the reactive program before the next event is considered. Constant-time (non-reactive) programming languages such as FaCT [10] and Jasmin [2] are concerned with bounding execution times of certain instructions to prevent leaking secret information (encryption keys) based on execution time. Constant-time programming is not a restriction of program complexity, but a restriction on the execution time of specific instructions, which may not vary in function of sensitive run-time



■ **Figure 2** Termination guarantees for different levels of reactivity.

■ **Listing 2** REScala reactive program with side-effects.

```

1 val counter = Var(0) // initial source value is 0
2 Signal { print("A" + counter() + " ") }
3 Signal { print("B" + counter() + " ") }

```

information. ActiveSheets [54] is a reactive language where programs consist of Microsoft Excel spreadsheets. While many spreadsheet operations are strongly reactive (e.g. arithmetic), there are exceptions such as SEARCH and REPLACE for searching and replacing substrings. Finally, the RT-FRP language [56] is a statically typed reactive programming language where the time and space (in memory) cost for each execution step is statically bound. To the best of our knowledge this language is strongly reactive.

2.1.4 Summary

Expressions in existing reactive languages and frameworks can accidentally or unintentionally cause long lasting computations that block the reactive program, thus turning the program no longer reactive. We identified 3 different levels of reactivity that are summarised in Figure 2, together with the possible techniques on how to enforce them. Stricter enforcement of program termination will result in reactive programs that have stronger guarantees with respect to the processing of their input, i.e. a more reliable upper bound can be placed on their execution time. However, the trade-off is that the types of programs that can be written is also reduced with stricter enforcement. It is up to developers to choose a reactive language or framework that can enforce a particular level of reactivity that is appropriate for the application at hand, i.e. eventually reactive or strongly reactive.

2.2 Embedding Imperative Code in Reactive Code

Effectful computations are extremely tricky to understand and debug when they are embedded within the nodes of a dependency graph [19, 33]. Consider the reactive program (written in REScala) in Listing 2. Line 1 defines a new source node called `counter`, and Lines 2 and 3 define two internal nodes that print either A or B to the console, followed by the value of the counter. When this program is executed, the initial value of `counter` is propagated through the program, and "A0 B0" is printed to the console in the order of evaluation (from top to bottom). However, when the value of `counter` is updated to 1, approximately 50% of the time the program output is reversed and "B1 A1" is printed. We call this problem the **Reactive Update Order Leak**, because effectful computations leak information about the update order of subexpressions within reactive programs, and their correct execution may even rely on a specific order.

The update order of the DAG is usually not part of the semantics of a reactive program. Reactive programming languages such as FrTime [12], Flapjax [34] and REScala [46] prevent *glitches* (temporary inconsistencies in the program [12]) by specifying that updates should be executed in a topological order of the dependency graph. Some implementations parallelise

19:6 Tackling the Awkward Squad for Reactive Programming

the execution of certain regions of the DAG, such as the conceptual propagation model of Elm [15] and a parallel version of the REScala update algorithm [18]. Streaming frameworks such as ReactiveX [40] and Akka Streams [45, Chapter 13] do not feature such an algorithm, and instead only specify that parent nodes should be updated before their child nodes.

All of the aforementioned technologies allow multiple valid update orders to be used for a given program. This is good for language implementers, because it gives them a lot of freedom to tweak and optimise (e.g. parallelise) how values propagate through the reactive program. However, for application developers this means that the concrete update order can vary across different implementations or versions of the same language or framework. The order can even change at run-time, which is the case in our experiments with the code snippet of Listing 2, where the execution of the program yields nondeterministic results.

The root of the problem are unconstrained side-effects in interactive applications. Not only can side-effects cause bugs that are difficult to find and reproduce because of an unlucky ordering in some propagations through the DAG, but they are also very difficult to coordinate and have a detrimental effect on behavioural composition [20]. Recognising these issues, most reactive programming languages and frameworks already forbid side-effects within the DAG of a reactive program, either through language design or via programmer guidelines (e.g. REScala guidelines [41, 1.5.3]). However, as we will discuss in Section 2.3, they are rarely successful in banishing side-effects completely.

The language design problem that needs to be solved is how to allow the coexistence of effectful computations that react to values arriving at a certain internal node of the dependency graph without accidentally or nondeterministically affecting the behaviour of effectful computations that reside in other nodes of the dependency graph.

2.3 Embedding Reactive Code in Imperative Code

Programs written in existing reactive programming languages and frameworks are not subject to the Reactive Update Order Leak (Section 2.2) when their code is *purely* functional. However, we observe that this requirement is not met in practice, because parts of real-world programs often depend on long lasting computations and effectful computations. For instance, input can come from a GUI or be streamed in from another computer over a network connection, and output may be used to modify a GUI or to push a notification to a user. A complete solution must therefore be able to bridge what we call the **Reactive/Imperative Impedance Mismatch**, in analogy with the Object-Relational Impedance Mismatch for object-oriented programming.

Because the embedding of reactive code within imperative code is unavoidable in real-world reactive applications, existing reactive languages and frameworks all tackle the Reactive/Imperative Impedance Mismatch to some degree. We argue that their solutions either have limited applicability, or are ad hoc solutions with unclear semantics. What follows is a non-exhaustive list of mechanisms that we could identify in related work.

Domain specific features are special language features tailored to a specific domain, usually involving GUIs. For instance, Flapjax [34], Elm [15], and ReactJS [28] provide a DSL for building GUIs whereby source nodes are automatically created and updated by GUI components, and the GUI automatically integrates with sink nodes. Such languages typically also feature built-in domain specific signals with narrow applicability, such as Elm's `Mouse.position` signal [15].

Special forms are built-in language constructs with special evaluation rules. Reactive languages often offer special forms to provide operators that cannot be implemented from within the language itself. For example, Elm [15] offers a built-in `syncGet` operation to execute synchronous HTTP requests. To prevent this computation from blocking the reactive program, Elm also offers an `async` special form to execute operations in the background.

Metaprogramming involves mechanisms to manually construct or modify parts of the reactive program. For example, FrTime and REScala offer built-in primitives to create and (destructively) modify source nodes of the reactive program. The semantics of manual assignments to source nodes are often unclear, especially when multiple threads of execution are involved. Streaming frameworks such as ReactiveX [40] and Akka Streams [45, Chapter 13] offer built-in operators to define new streams ex nihilo, reducing their dependence on special forms. Still, if there is no built-in support for a specific data source, a programmer has to open up the implementation of operators to carefully craft a new one.

Hidden concurrency involves mechanisms whereby multiple threads of execution are used, but where the concurrent nature of the operations is hidden from the programmer. For example, in FrTime a dedicated thread manages the execution of the reactive program behind the scenes, while the Racket Read-Eval-Print Loop continues running on the main thread. From the REPL, a Scheme program can “send” input values to the reactive thread. Conversely, the output of FrTime programs can be reactively displayed in the output of the REPL. However, multi-threading and concurrency-control are not part of the FrTime programming model.

Periodic polling enables embedding by periodically updating the source values of a reactive program from some computational process that is external to the reactive program. The quintessential example of this approach is the `seconds` behaviour as found in FrTime [12]. The `seconds` behaviour is a free variable in the reactive program that is automatically updated to the current Unix time. Typically, dependents of the `seconds` behaviour use the timestamp only to determine whether they should imperatively poll an application-specific data source. Such code lets the reactive runtime schedule imperative code, giving rise to Reactive Update Order Leaks.

The language design problem that needs to be solved is how to design a reactive programming language that supports the features of the awkward squad, but without introducing the Reactive Thread Hijacking Problem and the Reactive Update Order Leak.

In other words, it is important that imperative code and reactive code can coexist within the same application in such a way that the imperative code cannot accidentally violate the invariants of the reactive code. The mechanisms employed by existing languages and frameworks are often highly specialised, have unclear semantics with regards to their interactions with the reactive program, and they do not solve these problems.

2.4 Solution: General Idea

The idea is to embrace that reactive applications always consist of both imperative and reactive parts. Both are desired, and they complement each other [19]. We give each their own thread and effect-set, and design simple composition operators to link them together.

19:8 Tackling the Awkward Squad for Reactive Programming

■ **Listing 3** A “Hello World!” program in Stella.

```
1 (actor Main
2   (def-constructor (start)
3     (println "Hello World!"))))
```

The result is called the Actor-Reactor Model, which can be summarised as follows.

- *Actors* are used to represent the imperative input and output parts of reactive programs. They imperatively manage their own state and input-output, and are solely responsible for executing long lasting computations and effectful computations.
- *Reactors* are used to encapsulate reactive programs, each with their own DAG and update thread. By construction it is impossible for reactors to perform long lasting computations and side-effects.
- Coordination of actors and reactors happens via the *data streams* which they consume and produce. We define a set of composition operators whereby the streams defined by actors can be linked to the source nodes of reactors and vice-versa how actors can act on changes to the sink nodes of reactors.

We have implemented the Actor-Reactor Model in an experimental language called Stella, which can be roughly divided into two levels. First, an object-oriented base language is used to restrict reactors from producing side-effects and performing long lasting computations (Section 3). Second, actors and reactors are used to strictly separate the imperative and reactive parts of programs (Section 4). We have written a prototypical Continuation-Passing Style interpreter for Stella in TypeScript (in the style of [22, Chapter 5]).

3 Base Language: OOP with Effect and Termination Guarantees

In this section we will focus on the object-oriented base language of Stella. Throughout all code examples we will consistently syntax highlight special forms (expressions with special evaluation rules) in blue. Strings are surrounded by double quotes and are highlighted in green. Symbols start with a single quote and are highlighted in red.

Stella is a dynamically typed language where all run-time values are objects. Its native objects are booleans, numbers, strings, symbols and the value `#undefined`, which are instances of the classes `Boolean`, `Number`, `String`, `Symbol` and `Undefined` respectively. A program in Stella consists of 3 sets of top-level definitions: a set of classes, a set of *actor behaviours* (“the class of an actor”), and a set of *reactor behaviours* (“the class of a reactor”).

3.1 Basic Expressions and “Hello World!”

Each Stella program must contain an actor behaviour called `Main`. This actor behaviour must have a constructor named `start`. To start a Stella program, the Stella runtime spawns an instance of the `Main` actor behaviour and invokes the `start` constructor. Consider the “Hello World!” program in Listing 3 that defines such a behaviour. The body of its `start` constructor contains a single `println` statement in operator prefix notation (Polish Notation). This can be seen as synchronously sending a `println` message with no arguments to the “Hello World!” object of class `String`. Similarly, `(+ 1 2)` can be seen as sending a `+` message to the object representation of the number 1 with one argument which is the object corresponding to number 2. In the rest of this paper we will use the terminology of “calling” or “invoking” a method rather than sending synchronous messages (cf. SmallTalk [24]).

■ **Listing 4** Examples of basic expressions in Stella

```

1 (def hello "Hey")
2 (set! hello "... from the other side")
3 (if (equal? hello "Hey") (println "yes") (println "no")) // console prints "no"

```

Expressions that may be used in the body of constructors and methods follow S-expression syntax [1, Chapter 1] and are shown in Listing 4. Local variables are introduced via the `def` special form (similar to `define` in Scheme), assignments use the `set!` special form, conditionals use the `if` and `cond` special forms. Two methods to test for equality are implemented by the superclass of all classes (`Object`): `equal?` tests for object equality, `eq?` for object reference equality. All values are `#true` except for `#false` and `#undefined`.

3.2 Abstract Data Types

Code in Stella can be either imperative or reactive. Both kinds of code can operate on the same data, but the allowed sets of operations differ. Whereas imperative code may use the full power of a Turing-complete language, reactive code is restricted to operations that are guaranteed to terminate and that are free from side-effects. To this end, abstract data types in Stella are represented by classes which may define local fields, constructors, methods, as well as *routines*. Routines are special kinds of methods whose expressive power is a strict subset of that of methods. Routines have the following properties.

1. Routines have no side-effects.
2. Routines always terminate.
3. Routines can only invoke other routines.

To explain how we can enforce those properties, consider the `Pair` class defined in Listing 5 which can be used to represent linked lists. Local fields of the class are declared on Line 2. Line 4 defines a constructor called `initialize-with` with 2 formal parameters called `initial-car` and `initial-cdr` that will initialize the fields of a new pair. Lines 8–9 define two routines called `first` and `second` with no arguments which are “getters” for the `car` and `cdr` field, and correspondingly, Lines 10 and 11 define two methods `set-first!` and `set-second!` which are setters for these fields. Line 13 defines a routine called `length` that will compute the length of a linked list by calling `length` on the `cdr` field as long as it is also a pair¹. We can enforce the properties of routines as follows.

Routines that contain expressions with side-effects are rejected by the interpreter. In our case they are `set!` and a couple of other special forms². Together with property #3, which is upheld using a run-time check, this ensures that routines will never have side-effects. This must be checked at run-time because Stella is a dynamic language. A run-time error occurs when a routine calls a method, for example `println` which performs IO.

Our current implementation of Stella uses *size-change termination* (SCT) for higher-order programs [35] to ensure at run-time that routines terminate. In a nutshell, this form of SCT dynamically constructs a *size-change graph* based on the argument values of a routine call

¹ Note that the `type-of` invocation in Listing 5 Line 15 returns a symbol that represents the name of the class. This is because classes are not reified as objects in our language (such as in SmallTalk [24, Chapter 5]). They cannot be referenced directly *except* via the `new` special form to create an instance.

² The special forms forbidden in reactors and routines are `set!`, `spawn-actor`, `spawn-reactor`, `send`, `emit`, `monitor`, and `react-to`.

19:10 Tackling the Awkward Squad for Reactive Programming

■ **Listing 5** The Pair class which has 2 kinds of operations, methods and routines.

```
1 (class Pair
2   (def-fields car cdr)
3
4   (def-constructor (initialize-with initial-car initial-cdr)
5     (set! car initial-car)
6     (set! cdr initial-cdr))
7
8   (def-routine (first) car)
9   (def-routine (second) cdr)
10  (def-method (set-first! new-car) (set! car new-car))
11  (def-method (set-second! new-cdr) (set! cdr new-cdr))
12
13  (def-routine (length)
14    (cond ((eq? cdr #undefined) 1)
15          ((eq? (type-of cdr) 'Pair) (+ 1 (length cdr)))
16          (else 2))))
```

■ **Listing 6** Creating a circular data structure with Pair of Listing 5.

```
1 (def p1 (new Pair 'initialize-with 1 2))
2 (set-second! p1 p1)
3 (length p1) // successfully rejected via a run-time error
```

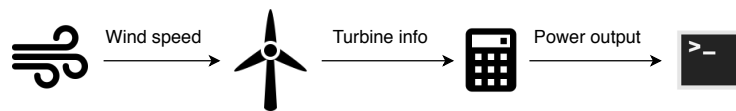
every time a routine is called. Before entering a new routine call, the *size-change termination principle* is used to compare the argument values to those of earlier calls to that routine (of the same class) higher on the call stack. If the new argument list is not decreasing in size, a run-time exception is thrown. Note that the algorithm assumes a well-founded partial order on values, for example, for numbers this is defined as $|x| < |y|$. In this case a recursive routine such as `factorial` is permitted as long as recursion stops when the value of its argument has decreased to 0.

As an example, consider the excerpt of Stella code in Listing 6 that uses the `Pair` class of Listing 5. Here, a `Pair` is made to point to itself, creating a circular linked list. A subsequent call to `length` on the `Pair` gives rise to a recursive call to `length` on the same class. SCT rejects this call, since the argument values have not decreased since previous invocations.

While [35] report that the overhead of their SCT analysis can be as high as a factor 100 (e.g. for merge-sort), it is negligible for applications that perform a lot of work in between recursive calls (they give a factorial function as an example). In any case, overhead remains constant with respect to the size of the input of the program. Crucially, it works for high-level programming languages such as Stella, and – unlike other approaches such as specialized type systems – SCT requires no programmer assistance. Stella is not coupled to any specific SCT algorithm, or even to SCT itself. What *is* part of the semantics of Stella, is that long-lasting computations are illegal, and that termination of routines must be enforced by the Stella runtime.

4 The Actor-Reactor Model

With the base language defined, we can now describe how Stella tackles the problem described in Section 2.3. We sketched the general idea in Section 2.4. The Actor-Reactor Model separates programs into *actors* and *reactors*. Actors handle the parts of a program that involve long-lasting computations or side-effects, whereas reactors handle the parts that are inherently reactive, or that are more easily expressed using reactive programming.



■ **Figure 3** Diagram of data flow in a wind turbine simulator consisting of a wind, a turbine, a turbine power calculator, and a console to print the result.

Coordination between actors and reactors is achieved using *data streams* that are produced by actors and reactors. Stella defines a number of composition operators to statically and dynamically manage how data can flow to and from actors and reactors. In the following sections we first introduce actors and data streams. In Section 4.3 we introduce reactors and how to compose them with actors.

4.1 Running Example: Wind Turbine Simulator

Consider a wind turbine simulator that calculates the real-time power output of a wind turbine. A simple simulator can be defined using the 4 components depicted in Figure 3. From left to right: a blowing wind, a wind turbine, a power calculator, and a console to print the output. Simulating wind can involve complex wind patterns and how they affect a turbine, or it can be a simple process that generates random values corresponding to the current wind speed. Even the simple version is challenging because it involves a process “a wind” that is continuously running and always changing. It also requires reactive processes: A blowing wind impacts the rotation of the wind turbine, which then impacts the power output. Important to note is that eliminating long lasting computations within reactive programs by itself is not enough to prevent them from blocking reactive programs. It is equally important that reactive processes cannot be accidentally blocked by long lasting computations in other parts of the program, for example the simulation of wind patterns.

We model wind as an actor because it is inherently an active process that is always running and changing independently. The console is also an actor because it performs IO. We opt to model a turbine and the power calculator as reactors: a turbine *reacts* to a wind, and a power calculator *reacts* to changes in the turbine.

4.2 Actors and Data Streams

Actors are typically defined in terms of a *behaviour* and a *mailbox* [30]. The behaviour of an actor describes its internal state and the messages that can be processed. Messages that are sent to an actor are inserted into its mailbox (e.g. a FIFO queue), and the actor continuously dequeues messages from its mailbox to process them one-by-one. Actors in Stella are based on the Active Objects model [57, 30, 16], where actor behaviours are defined similar to classes in object-oriented programming. In addition to receiving and processing messages, actors can be used to implement zero or more data streams to which they can *emit* (publish) values. The following sections explain the different parts of actor behaviours using the wind from our running example. We explain how actor behaviours are defined, how data streams can be implemented using actors, and how actors can monitor data streams.

4.2.1 Actor Behaviours

Listing 7 defines the `Wind` actor behaviour. An actor behaviour has a number of local fields, in this case 1 field called `rng` (Line 3). A constructor called `init` is defined on Line 5, which initializes the `rng` field with a new random number generator (an object). A method called

19:12 Tackling the Awkward Squad for Reactive Programming

■ **Listing 7** The `Wind` actor behaviour to implement a stream that represents wind speed.

```
1 (actor Wind
2   (def-stream speed 1)
3   (def-fields rng)
4
5   (def-constructor (init) (set! rng (new Random))))
6
7   (def-method (blow)
8     (emit speed (integer-between rng 0 30))
9     (sleep 10000)
10    (send #self 'blow)))
```

`blow` without arguments is defined on Line 7, whose implementation we will explain later when we define data streams. A `Wind` actor is thus capable of processing `blow` messages that are inserted into its mailbox, which amounts to invoking the corresponding `blow` method.

The special form `spawn-actor` is used to spawn new actors, in this case to create new winds. The following expression spawns an instance of `Wind`, which could be used to represent the mistral wind. A reference is returned to the new actor, which is an object of type `ActorReference`. To initialize the actor, `spawn-actor` takes the name of a constructor (as a symbol) and any arguments it requires. Constructors are special kinds of methods that may only be invoked once, and only as the very first message an actor processes. The act of spawning an actor via a constructor is semantically equivalent to spawning an actor and inserting a message in its empty mailbox that will initialize the actor.

```
1 (def mistral (spawn-actor Wind 'init))
```

Actors communicate via asynchronous message passing via the `send` special form that inserts a new message in the mailbox of the designated actor. The following expression sends a `blow` message to `mistral`, which is expected to be an actor reference. While in this case `blow` does not expect any arguments, they would be provided after the `'blow` symbol. The message payload between actors (and reactors) is always passed by (deep) copy, and actors can send messages to themselves by sending them to `#self`, which represents a reference to the current actor.

```
1 (send mistral 'blow)
```

4.2.2 Declaring Data Streams

Every actor can export data streams. The behaviour of an actor determines which data streams an actor implements via `def-stream` as seen on Line 2 of Listing 7. This expression takes two arguments: the name of the stream and its *arity*. The name is used to uniquely identify a particular stream that belongs to a given actor, and the arity of a stream specifies the number of elements that must be emitted to the stream in one “emit step”. In our example a `Wind` actor exports a single stream called `speed` with arity 1.

Stream arity is used to emit new values that are intrinsically connected, and must therefore always change simultaneously. For example, an actor might export a stream called `location` of arity 2 that represents coordinates in the form of `latitude` and `longitude`. Since they describe the real-time location of some real-world moving object, latitude and longitude should always change simultaneously. Otherwise, a consumer of the stream might first update the entire application (e.g. some world map) with a new value for latitude, and only after some time with the corresponding value for longitude. After the first update the

Listing 8 Monitoring data streams with actors

```
1 (actor Main
2   (def-constructor (start)
3     (def sirocco (spawn-actor Wind 'init))
4     (monitor sirocco.speed 'print-wind))
5
6   (def-method (print-wind wind-speed)
7     (println "the new wind speed is: " wind-speed)))
```

application would be in an inconsistent state, similar to a *glitch* in reactive programming [12]. An alternative approach is to emit `location` as a single object. However, we will use stream arity to facilitate the composition of actors and reactors in Section 4.3.

4.2.3 Publishing to Data Streams

Actors can emit values to their own data streams by using the `emit` special form. Emitting a value amounts to sending the new value to all subscribers of the stream. Consider the `blow` method in Listing 7. Whenever a `Wind` actor processes a `blow` message, on Line 8 the actor will `emit` a new value to the `speed` stream, which in this case is a random number between 0 and 30 representing the wind speed in meters/second. Now, a special type of message (a publication) is added to the mailbox of subscribers, which are other actors or reactors. Because the `speed` stream is defined with arity 1, `emit` only requires 1 argument. The actor then sleeps for 10000ms (Line 9), and afterwards sends itself a new `blow` message to emit another value (Line 10).

4.2.4 Qualifying and Monitoring Data Streams

Two mechanisms are required to create a subscription on a data stream: *qualification* and *monitoring*. Both are explained using the `Main` behaviour in Listing 8 which can be seen as the console from our running example, but instead of monitoring and printing power output, it monitors and prints the wind speed. When the `start` constructor is executed, Line 3 first spawns an instance of the `Wind` behaviour, and Line 4 exemplifies both qualification and monitoring.

Qualification is the act of designating a reference to a particular stream exported by a particular actor (or reactor). On Line 4, the expression `sirocco.speed` evaluates to an object of class `Stream` that represents a reference to the `speed` stream exported by the `sirocco` actor. Data will only start flowing once a consumer (an actor or reactor) subscribes to the stream, in which case data flows directly from the producer to the consumer.

Actors subscribe to data streams by monitoring them. This is exemplified by Line 4, where the `sirocco.speed` stream is monitored for changes. Whenever this stream emits a new value, a `'print-wind` message will enter the mailbox of the actor, which is processed by the corresponding `print-wind` method on Line 6. This method requires exactly 1 formal parameter because the `speed` stream has an arity of 1. We will see in Section 4.3 that reactors do not explicitly monitor data streams like actors, and instead they will automatically react to values that are emitted to data streams.

4.3 Reactors

Reactive languages rely on 2 fundamental mechanisms. First, at compile-time, the program text is compiled to a DAG that consists of source nodes that represent the input of the reactive program, sink nodes represent the output of the reactive program, and internal nodes represent all computations that occur between the sources and sinks. Second, at run-time, a *reactive engine* is responsible for propagating new values through the DAG, such that the calculation that makes up the output remains consistent with the values of the input. The reactive engine is smart enough to prevent glitches [12].

In the following sections we gradually explain Stella’s reactors by implementing a simple reactive wind turbine and power calculator. Stella features 2 different ways to statically compose *reactor behaviours*: via *point-wise* and *point-free* composition, named after function composition in Haskell [32, Chapter 5]. In Section 4.3.6 we explain the run-time semantics of reactors and how they manage dependencies on data streams.

4.3.1 Definitions

A reactor consists of 3 layers.

Layer 1: Reactor Behaviour. A *reactor behaviour* describes the static properties of a reactive program, represented by a DAG that is constructed at “DAG compile-time” (a pre-processing step of our interpreter). The DAG is constructed from the program text, for example the reactor behaviour of Listing 9 which we will explain in Section 4.3.2. It describes the source nodes, sink nodes, and internal nodes of the reactive program, and all of the dependencies between them. We may refer to an actor or reactor behaviour as simply “behaviour” if it is clear from context to which one we are referring.

Layer 2: Reactor Deployment. Reactive languages usually store the run-time information of a reactive program (e.g. node values and local state) directly in the nodes of the DAG. However, in our case reactor behaviours can be composed and reused by multiple reactors, and every use of a DAG can be in a different state depending on the values that were propagated. Therefore, a *reactor deployment* represents a specific instance of a reactor behaviour. In other words, a reactor deployment stores all run-time information pertinent to a specific instance of a DAG that is used by a specific reactor.

Layer 3: Reactor. A *reactor* is process with a mailbox and a *vat* (collection) of reactor deployments. It is the driving force behind a reactive program: a reactor continuously dequeues values from its mailbox and propagates them through the destined deployment via a built-in reactive engine. Similar to how actors are spawned from actor behaviours, a reactor is spawned from a reactor behaviour (that represents a DAG). At spawn-time, the reactor creates an initial deployment for this DAG, which we call the *root deployment*. A reactor has exactly 1 output stream of arity n , where n corresponds to the number of sinks of the root deployment. Every time the root deployment updates, its sink values are emitted on the output stream of the reactor. Our definition of reactors intentionally covers deployments that give rise to other deployments within the same vat, but we will not discuss those features in this paper. Reactors will therefore always contain exactly 1 deployment (namely the root deployment).

4.3.2 Basic Reactor Behaviours

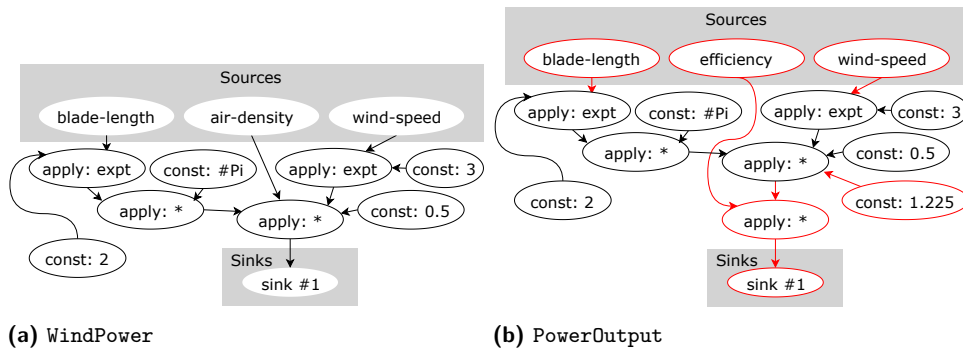
A reactor behaviour is the textual representation of the DAG of a reactive program: it has a name, at least 1 source, at least 1 sink, and any number of internal nodes that describe the computations between the sources and sinks. An example of a computation is the theoretical

■ **Listing 9** The `WindPower` behaviour calculates the maximum theoretical power output of a turbine.

```

1 (reactor (WindPower blade-length air-density wind-speed)
2   (def swept-area (* #Pi (expt blade-length 2)))
3   (out (* 0.5 swept-area air-density (expt wind-speed 3))))

```



■ **Figure 4** Side-by-side comparison for the DAGs of `WindPower` and `PowerOutput` of Listing 9 and 10. Nodes and dependencies introduced by `PowerOutput` are highlighted in red.

power output of a wind turbine (in Watt) which is based on the area swept by its blades, the air density surrounding the turbine, and the velocity of the wind [43]. It can be calculated as follows:

$$Power (W) = 0.5 \times Swept Area (m^2) \times Air Density (kg/m^3) \times Velocity^3 (m/s)$$

This formula is implemented in Listing 9 in a reactor behaviour called `WindPower` with 3 source nodes and 1 sink node. Its 3 sources are called `blade-length`, `air-density`, and `wind-speed`. There is one local variable called `swept-area`, and one sink node that is immediately linked to the result of a multiplication. The DAG of this behaviour is depicted in Figure 4a. Every invocation of a routine is depicted by an “apply” node, and all expressions without dependencies are wrapped in a “const” node that will be computed when the DAG is compiled. Since the run-time values that are propagated through the DAG are regular objects, reactors can only invoke routines on those objects, and the invocation of regular methods will result in a run-time error.

4.3.3 Point-wise Graph Composition

While the `WindPower` behaviour computes the theoretical power output of a turbine, a more accurate calculation takes into account turbine efficiency (typically between 10–30% [42]). To this end, Listing 10 defines a behaviour called `PowerOutput` that shows how reactor behaviours can be composed in a point-wise manner. It has 3 sources called `blade-length`, `efficiency` and `wind-speed`. They correspond to the 3 pieces of information that a wind

■ **Listing 10** Point-wise composition of reactor behaviours.

```

1 (reactor (PowerOutput blade-length efficiency wind-speed)
2   (def wind-power (tick WindPower blade-length 1.225 wind-speed))
3   (out (* efficiency wind-power)))

```

19:16 Tackling the Awkward Squad for Reactive Programming

■ **Listing 11** The Turbine behaviour implements a simple wind turbine.

```
1 (reactor (Turbine blade-length efficiency wind)
2   (out blade-length efficiency wind.speed))
```

turbine is expected to produce to be able to calculate its power output. Line 2 performs a point-wise composition of reactor behaviours via the `tick` special form. This can be thought of as a function application but for reactor behaviours, of which the result is bound to the `wind-power` variable. Line 3 then scales the theoretical output by the efficiency of the turbine.

A `tick` operation is resolved at compile-time as the inlining of a DAG. First, the source nodes of the composed behaviour (`WindPower`) are connected to the corresponding arguments of `tick`. Second, the sink node of the composed behaviour is connected back into the composer, in this case to all nodes in `PowerOutput` that use the `wind-power` variable (multiple sinks would be defined via a `def-values` special form). The resulting graph is depicted in Figure 4b, where the nodes and dependencies introduced by `PowerOutput` are highlighted in red. For brevity, in the `tick` expression we assume a default value of 1.225kg/m^3 (the air density at 15°C at sea level).

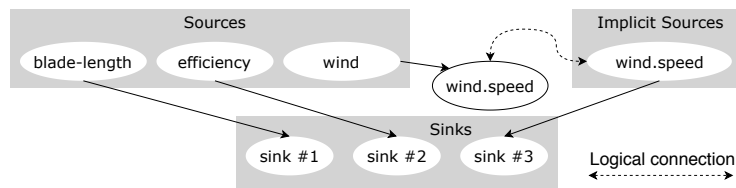
While it is not part of our goals, the structure of the DAG can be optimised when it is compiled. In this case the source and sink nodes of the inlined `WindPower` behaviour were automatically eliminated because they are redundant, and because (by definition) sources and sinks can only exist at the boundaries of a DAG. For example, the constant value 1.225 provided by the composer in Listing 10 is represented by a constant node in Figure 4b (depicted in the bottom right). This node is directly connected to the apply node because the original `air-density` source node (which it replaces) was eliminated.

4.3.4 Behaviour Stream Composition

From the `PowerOutput` behaviour we know that a wind turbine should provide its blade length, efficiency, and the current wind speed affecting it. The simplest possible implementation of a turbine that we can think of is defined in Listing 11. This reactor behaviour has 3 sources: `blade-length` is a number usually between 20 and 80 (meters), `efficiency` is a number between 0.1 and 0.3 (10-30%), and `wind` should be a reference to an actor. The output of `Turbine` is a stream of arity 3 that echoes the blade length and efficiency, and the qualification “`wind.speed`” will echo the contents of the `speed` stream.

A qualification in the body of a reactor behaviour represents a dependency to a stream. However, handling such dependencies can be quite tricky since there are potentially two kinds of changing values. First, the exporting actor of a stream may change, for instance when a new actor reference is propagated for the `wind` source node (e.g. `sirocco` or `mistral`). Second, the `speed` stream is continuously emitting new values. A reader may recognise this as a *higher-order* stream [34]. The source node is conceptually a stream of values, and every value is an actor (or reactor) that exports streams.

Inspired by the compilation of the *async* expression in Elm [15], every qualification is compiled to 2 graph nodes. First, an internal node manages the dependency on the referenced stream. Second, an *implicit source node* is responsible for processing the values emitted by the stream. The resulting graph for `Turbine` is depicted in Figure 5. When the source node `wind` changes to a different actor (e.g. another wind) then this new value is propagated to the special qualification node. This node unsubscribes from its current stream (if present)



■ **Figure 5** DAG of the Turbine behaviour.

■ **Listing 12** Point-free composition of reactor behaviours.

```
1 (reactor TurbinePowerOutput (ror PowerOutput Turbine))
```

and subscribes to the newly referenced `speed` stream. The value of the implicit source is immediately changed to the most recent value emitted by the newly referenced stream. From that point onwards, whenever publications of the new stream enter the mailbox of the reactor, the reactor will process them by changing the value of the corresponding implicit source node.

4.3.5 Point-free Graph Composition

There are two ways to implement a wind turbine that is linked to a power calculator. Either a new reactor is spawned for each of them and their input/output streams are subsequently linked together, or the 2 reactor behaviours are composed and spawned as a single reactor. Both approaches are valid, and which one is more desirable depends on the application. We take the second approach by composing the 2 behaviours via *point-free* graph composition.

In Haskell, new functions can be defined point-free via a function composition operator. The composition $f \circ g$ (“ f after g ”) is a function that first applies g to its argument, then f to the value returned by g ; $(f \circ g) x = f(g(x))$. Similarly, the `ror` operator composes reactor behaviours: $r_1 \circ r_2$ constructs a new behaviour where data is first propagated through r_2 and then through r_1 . The DAG of this behaviour is the composition of r_1 and r_2 , where the sinks of r_2 are connected to the sources of r_1 .

As an example, Listing 12 defines a new behaviour called `TurbinePowerOutput` that combines the behaviours of `Turbine` and `PowerOutput`. We designed those behaviours such that they can be easily composed, i.e. the sinks of `Turbine` directly match the sources of `PowerOutput`. If the behaviours would not directly fit together, intermediate behaviours can take care of reordering sources and sinks, or transforming data.

The `ror` operator is capable of connecting *multiple* “input” behaviours to 1 “output” behaviour. The following expression defines a new behaviour `R` that is the composition of an output behaviour R_{out} with input behaviours R_1 to R_n .

```
1 (reactor R (ror R_out R_1 R_2 ... R_n))
```

Behaviour `R` can be compiled as long as the number of sinks of all input behaviours matches the number of sources in R_{out} . If this is the case, they will be connected in order from left to right to construct the behaviour `R`. The sources of `R` will be the same as the sources of the inputs, ordered from left to right.

$$sources(R) := sources(R_1) + sources(R_2) + \dots + sources(R_n)$$

19:18 Tackling the Awkward Squad for Reactive Programming

The sinks of R are the same as the sinks of R_{out} .

$$\text{sinks}(R) := \text{sinks}(R_{out})$$

Additional point-free graph composition operators with different semantics are conceivable, such as the `parallel` and `parallel*` operators in [36].

4.3.6 Run-time Semantics of Reactors: Spawning and Linking

We now complete the running example of Section 4.1 by showing how the program is started, and how actors and reactors are linked together. We will focus on the composition of actors and reactors rather than the internal semantics of reactors. Internally, it suffices to know that every reactor has a reactive engine that is responsible for propagating values from sources to sinks. Similar to Flapjax [34] and REScala [46], we based our propagation algorithm on that of FrTime [12], but without the complexity of a dynamic dependency graph. It ensures that only the parts of a DAG that are affected by a change will be recomputed, and that computations produce no glitches when multiple nodes change simultaneously.

Listing 13 implements the `Main` program for the running example. Its purpose is to spawn an actor representing a wind, spawn a reactor representing a wind turbine and its power output calculation, and to print this power output to the console. This functionality is implemented by the `start` constructor. Most of the expressions have been discussed previously. Line 3 spawns an instance of the `Wind` actor behaviour and Line 4 sends it a `blow` message. The wind will now start periodically emitting values.

The `spawn-reactor` expression on Line 5 spawns a new reactor that is now waiting for data on its sources. Remember that a reactor is defined as a vat of reactor deployments, and in this case `TurbinePowerOutput` will be the root deployment, as well as the only deployment for this reactor throughout its lifetime.

Reactors are linked to actors (or other reactors) via the `react-to` special form that will change the values of source nodes. If the new value of a source node is an object of type `Stream`, instead of changing the value of the source node, the reactor will automatically create a subscription on the stream (possibly replacing an existing subscription). Then, whenever new values are emitted by this stream, they enter the mailbox of the reactor which will process them by modifying the value of the corresponding source node. In Listing 13 the sources of `turbine` are changed on Line 6 to `80`, `0.3`, and `sirocco`. In the context of our application, this means that this is a reactor that represents a turbine with a blade length of 80 meters, an efficiency of 30%, which is influenced by the `sirocco` wind. To print the output of the turbine the console, on Line 7 the main actor monitors the turbine for changes. Reactors export exactly 1 output stream called `out`.

Reactors are aware of stream arity, and the `react-to` composition operator can be used to react to streams with an arity greater than 1. In this case, the reactor requires exactly as many source nodes as the arity of the input streams. For example, a reactor that is made to react (via `react-to`) to a stream of arity 2 will require exactly 2 source nodes. Whenever the input stream emits new values they will enter the mailbox as a single publication, but (in this case) the value of the 2 source nodes of the reactor will change simultaneously. The `react-to` composition operator ensures that there is a one-to-one mapping between its arguments and the source nodes of the reactor.

■ **Listing 13** The Main program for the wind turbine simulator of Section 4.1.

```

1 (actor Main
2   (def-constructor (start)
3     (def sirocco (spawn-actor Wind 'init))
4     (send sirocco 'blow)
5     (def turbine (spawn-reactor TurbinePowerOutput))
6     (react-to turbine 80 0.3 sirocco)
7     (monitor turbine.out 'print))
8
9   (def-method (print watt)
10    (println "turbine produced: " (round (/ watt 1000000)) " MW")))

```

5 Evaluating the Awkward Squad for Reactive Programming

We introduced the awkward squad for reactive programming as a set of issues that are essential for real-world software development, but that do not fit within reactive programming. In this section we investigate the extent to which these issues can be present in existing reactive languages and frameworks. We find that it is indeed the case that existing languages and frameworks expose operations or mechanisms to developers that fall within one or all issues of the awkward squad. In many cases these operations and mechanisms will be unavoidable for developers, either because they are an inherent part of the language or framework, or because otherwise it would be impossible to write certain applications, e.g. applications with a GUI. In this paper we do not investigate the extent to which possibly issue-causing operations are used in real applications, and if they are present, which bugs they can possibly cause in those specific applications.

Table 1 lists a number of reactive languages and frameworks that we consider to be representative for the state-of-the-art. They are FrTime [12] (Racket), Flapjax [34] (JavaScript), REScala [46] (Scala), ReactJS [28] (JavaScript), Akka Streams [45, Chapter 13] (Scala), and RxJS [49] (JavaScript). We used them to implement our running example of Section 4.1³. Below the double horizontal line we list 3 other reactive languages which we did not use to implement the application (for technical reasons) but which we classified according to their respective papers. They will be discussed in Section 5.8. Based on our findings we categorised these languages and frameworks as follows.

Reactive Thread Hijacking Problem (RTHP). Does the language or framework solve the Reactive Thread Hijacking Problem? In other words, does the language or framework prevent infinite computations from blocking the reactive program? We make no distinction between eventual reactivity and strong reactivity. If not, we discuss the features that can be used to block the reactive program.

Reactive Update Order Leak (RUOL). Does the language or framework solve the Reactive Update Order Leak? In other words, does the language or framework disallow side-effects in internal nodes of the DAG? If not, we will highlight the features where side-effects can be executed inside the DAG.

Reactive/Imperative Impedance Mismatch (RIIM). In Section 2.3 we listed a number of different mechanisms used by reactive languages to be able to embed reactive code within imperative code. We discuss, to the best of our knowledge, which of the listed mechanisms are used.

³ All code from this paper and the implementations used to guide our evaluation of the different languages and frameworks are available in an artefact. To download this artefact, see the supplementary material on the first page of this paper.

19:20 Tackling the Awkward Squad for Reactive Programming

■ **Table 1** Categorisation of reactive languages and frameworks.

	RTHP	RUOL	RIIM
FrTime	×	×	Periodic polling, hidden concurrency, domain specific features, metaprogramming
Flapjax	×	×	Metaprogramming, domain specific features
REScala	×	×	Metaprogramming
ReactJS	×	×	Metaprogramming, domain specific features
Akka Streams	×	×	✓
RxJS	×	×	Metaprogramming
Stella	✓	✓	✓
Elm	×	×	Domain specific features, special forms
ActiveSheets	✓	✓	✓
Coherence	×	✓	✓

5.1 FrTime

FrTime is a functional reactive programming language built in Racket that can be interacted with via the Racket Read-Eval-Print Loop [12].

RTHP. The execution time of expressions in FrTime is unrestricted. While there exists a limited set of built-in functions (e.g. arithmetic) that always terminate, the built-in `lift-strict` primitive is used to integrate any (possibly infinitely looping) Racket function with the DAG. The implementation of FrTime exposes multiple threads of execution to developers: one thread is responsible for updating the reactive program, and another thread is responsible for the REPL. To implement an infinite loop that represents a wind, we blocked one thread of execution to continuously modify a source node of the dependency graph.

RUOL. Any Racket function may be used in internal nodes of the DAG via the aforementioned `lift-strict` function. Additionally, FrTime offers abstractions for “event streams” that can be mapped and filtered (via `map-e` and `filter-e` respectively) using regular Racket functions.

RIIM. We have identified 4 mechanisms that are used to embed reactive code within imperative code. Firstly, there are 2 built-in signals called `seconds` and `milliseconds` that are updated automatically by the runtime, which can be used for periodic polling. Secondly, multiple threads of execution are exposed to developers. While the Racket REPL thread can be used to send and receive values to and from the reactive program, the semantics of their interaction is not part of the language definition. Thirdly, FrTime has domain specific features in the form of a wrapper around the Racket GUI toolkit [26], which automatically integrates with the DAG. Lastly, source nodes of the DAG can be manually defined via `cells` and `event-receivers`, and they can be assigned to via `set-cell!` and `send-event` respectively. The semantics of modifying a source node is unspecified, especially when assignments to the same sources occur in multiple threads.

5.2 Flapjax

Flapjax is a reactive programming language based on JavaScript [34].

RTHP. Nodes in the dependency graph consist of arbitrary JavaScript functions, which are unrestricted in their execution time. To create an infinite loop that implements a wind without blocking the browser, we implemented a non-blocking loop by wrapping

JavaScript's `setTimeout` to asynchronously schedule an event in the JavaScript event-loop. Some built-in operations can unintentionally block the reactive program. For example, the `evalForeignScriptValE` operation is applied to a reactive value that contains a URL. It retrieves and evaluates the (foreign) JavaScript file on the URL, and publishes its return value as a new reactive value [50].

RUOL. Flapjax programs consist of arbitrary JavaScript expressions that may involve arbitrary side-effects inside the DAG. Some built-in operations execute side-effects in the DAG, such as the GUI modification operators `insertDomB` and `insertDomE` to insert a reactive value in the browser DOM, operations such as `getWebServiceObjectE` to perform XMLHttpRequests, or the aforementioned `evalForeignScriptValE` to evaluate an arbitrary JavaScript file.

RIIM. To embed reactive code within imperative JavaScript code, Flapjax has 2 mechanisms. Source nodes of the DAG can be manually created from the GUI via operations such as `extractValueB` and `extractValueE` which are updated automatically by the runtime, and ex nihilo via `receiverE` (to create a new event stream) and `sendEvent` (to modify an event stream). Flapjax also offers special features to construct GUI elements that automatically integrate with reactive values.

5.3 REScala

REScala is a reactive programming library in Scala that unifies the concepts of functional reactive programming with object-oriented programming [46].

RTHP. REScala imposes no restrictions on the execution time of expressions inside the DAG of a reactive program. Since it is built as a library, regular Scala functions are used to perform computations on reactive values. To model a wind from our running example without blocking the reactive program, we manually constructed a new Scala thread with an infinite loop that non-reactively modifies a source node of the reactive program.

RUOL. Since REScala is conceived as a library, the Scala functions used in nodes of the DAG may perform arbitrary side-effects. As a design guideline, the REScala documentation explicitly mentions that functions inside the DAG must be pure [41, 1.5.3].

RIIM. To embed reactive code within imperative code, REScala offers special features to manually create new source nodes of the DAG (`Vars`) and to modify them via assignment. Conversely, callbacks can be installed on sink nodes of the DAG to act on their changes.

5.4 ReactJS

ReactJS is a JavaScript reactive GUI framework developed by Facebook [28], which is used to develop reactive web applications and mobile applications [21].

RTHP. The types of expressions that constitute a ReactJS program are regular JavaScript expressions, and are unrestricted in their computation time. To implement a wind without blocking the browser, we used JavaScript's `setInterval` that calls a function on a repeated interval.

RUOL. There are no restrictions on side-effects inside a ReactJS dependency graph. The documentation mentions that reactive components should be pure only with respect to their "props" object, which is a framework-provided object that is used to create dependencies between reactive components [27].

19:22 Tackling the Awkward Squad for Reactive Programming

RIIM. The state of a reactive component is manually modified via a special `setState` method that triggers a new propagation cycle. ReactJS offers domain specific features (via *JSX templates*) to construct user interfaces that automatically display the values of sink nodes of the dependency graph.

5.5 Akka Streams

Akka Streams is a streaming library in Scala based on the Akka actor library [45, Chapter 13]. We had some difficulties reproducing the semantics of our running example in Akka Streams because we found it to be very difficult to create dependency graphs that are not linear (multiple sources or sinks) and which have similar update semantics.

RTHP. There are no restrictions on performing long lasting computations inside the DAG, e.g. via the stream operator `map` to apply a regular Scala function to a stream, or `zipWith` to combine 2 or more streams via a regular Scala function.

RUOL. Many built-in streaming operators are designed to be without side-effects. While we could not find explicit programmer guidelines discussing side-effects in the Akka Streams documentation, we believe that stream operators are intended to be pure, including those that accept arbitrary Scala functions (e.g. `map` or `zipWith`). We found at least some interest by users of the Akka project on GitHub for stream operators that execute side-effects. In one instance a user requested a novel operator to execute side-effects without performing a value transformation, noting that the only way to achieve the desired effect was via the `map` operator (which, in the experience of the user, lead to code duplication) [52]. In response to this issue a new `wireTap` stream operator was added to Akka version 2.5.13. A new variant of this operator (with different semantics) is currently being requested by a different user [53]. This anecdotal evidence is at least an indication that functional purity is not always upheld by the users of Akka. Some operators with side-effects are built-in, for example, the `log` operator logs the elements flowing through a stream, and the `ask` operator sends an asynchronous message to an actor. Sink nodes also perform side-effects to act on the elements of a data stream, e.g. to print results to the console via a `foreach` operator.

RIIM. As far as we know, Akka Streams has a clean separation between the code that is responsible for supplying values to the reactive program (which are actors) and the code that is responsible for the reactive program itself (which are actors that run data streams). Note that this is not always the case, since there exist many operators to create new source nodes ex nihilo whereby the reactive program (an actor running data streams) is itself responsible for collecting/retrieving its input data. For example, the `FilIO.fromPath` operator that creates a source node to read the contents of a file.

5.6 RxJS

RxJS [49] is a streaming library for JavaScript based on the ReactiveX specification [40], which has currently been implemented in 18 languages.

RTHP. There are no restrictions on the execution time of the expressions that constitute a stream, and long lasting computations will block the entire program. Implementations of ReactiveX in other languages may support *Schedulers* which are designed to introduce multithreaded processing to streams, but this can cause other issues, especially when used in combination with side-effects.

RUOL. Many built-in streaming operators are designed to be without side-effects, and the documentation of RxJS describes operators in general as “pure functions” [39]. We believe that operators that rely on arbitrary JavaScript functions (e.g. `map` or `zipWith`) are intended to be functionally pure as well (but which cannot be enforced). In some cases side-effects are unavoidable, for example when defining a type of source node called a `Subject`, which is manually updated to start the propagation of new values. Frameworks in the family of ReactiveX (such as RxJS) also offer a whole range of “Do” operators that are specifically designed to execute side-effects within a reactive program without performing a value transformation [38].

RIIM. Programmers in RxJS can manually create and update new source nodes (a `Subject`) of the DAG, and they can manually create new streaming operators using metaprogramming to read from unsupported data sources. Callbacks are registered on sink nodes of the program to imperatively act upon their changes (e.g. by modifying the GUI, or printing to the console).

5.7 Stella

RTHP. Stella solves the Reactive Thread Hijacking Problem by eliminating infinite computations from reactive programs. Whether our implementation also solves the problem of responsiveness in general depends on the expectations of the application developer. We believe there is no one-size-fits-all solution to ensure that an application remains “reactive” or “responsive”, since their meaning is likely to change depending on the application requirements or domain. The Actor-Reactor Model facilitates restricting certain parts of the application (the reactive parts) to provide extra guarantees with respect to responsiveness or computational complexity without introducing the problems that we identify in this paper. The design choice that we made in Stella is to enforce eventual reactivity via size-change termination. Thus, Stella ensures that reactors must *eventually* terminate, and that the execution thread of a reactor is not accidentally hijacked by computations in other parts of the program (other actors or reactors). In different application domains with stronger memory or timing requirements (e.g. safety systems, robotics, ...) it would be conceivable to further restrict reactors, for example by imposing strong reactivity and bounded-size mailboxes.

RUOL. Stella solves the Reactive Update Order Leak by ensuring that effectful computations cannot be part of the dependency graph of a reactive program. This is realised by routines in the object-oriented base language. However, using methods and routines may be a burden for programmers, since they must make an effort to correctly program a piece of functionality as a regular method or as a routine. When using built-in classes or libraries they must also know whether functionality is offered as a method or as a routine.

RIIM. Stella solves the Reactive/Imperative Impedance Mismatch. It ensures that the embedding of reactive code within imperative code does not introduce the Reactive Thread Hijacking Problem and the Reactive Update Order Leak, and that the semantics of embedded reactive code are clear. There are 2 composition operators available to actors: `react-to` and `monitor`. To imperatively change the source nodes of a reactive program, actors must use `react-to` with clearly defined semantics: a message is sent to the reactor that, when processed, changes its source nodes. To imperatively act on the changes of reactive programs, actors must use `monitor`: whenever the monitored stream produces a new result, a new message is enqueued in the mailbox of the actor. Reactors have no imperative operators to “send” or “receive” values, or to manually react to the changes of values (e.g. via callbacks). In the true spirit of reactive programming, they can only declaratively express dependencies on data sources via their source nodes and qualifications.

5.8 Additional Mentions

There are some reactive languages and frameworks that require a special mention. When possible we list them in Table 1 below the double horizontal line.

Elm [15] is a reactive programming language that compiles to JavaScript. We were unable to build our running example in Elm because its current distribution is no longer reactive [14]. Expressions inside the DAG in Elm may perform infinite computations that block the reactive program. An interesting observation is that Elm is presented as a purely functional reactive programming language, but its paper describes a `syncGet` operation to execute a web request (e.g. to fetch an image from a URL), which is clearly a side-effect. The reason why this operation (among others) is built-in is because it is *necessary* for building web applications, but introducing this operation in the reactive language also causes the problems of the awkward squad. This is exactly the essence of the awkward squad.

ActiveSheets [54] is a reactive language based on Microsoft Excel where the DAG consists of regular spreadsheet operations. ActiveSheets adds features to Excel to automatically insert values into cells based on external data streams, and, like a regular spreadsheet, updates to cells automatically propagate throughout the program. The core language of ActiveSheets is formalised and used to prove that, for any given update of a cell, computation time and memory usage are bound. Furthermore, as far as we know there are no spreadsheet operations that have side-effects on other cells, so there can be no side-effects in internal nodes of the DAG. However, ActiveSheets is not a general purpose programming language. Conceptually an ActiveSheets program can be represented by 1 reactor that implements the spreadsheet logic.

In the Coherence language, code is divided in *derivations* and *reactions* [19]. Derivation is used to automatically compute the program output by deriving values from input via purely functional computations. Side-effects are isolated to different parts of the code, namely reactions, that are responsible for imperatively keeping the application state consistent with derived values. This stems from the insight that derivation and reaction need each other, but that coordinating side-effects in an event-based application is extremely difficult. A reactive program constructed via derivations is guaranteed to be free of side-effects, and is thus not subject to the Reactive Update Order Leak. The Coherent Reaction programming model offers no mechanisms to solve the Reactive Thread Hijacking Problem.

HipHop is a synchronous reactive programming language inspired by Esterel with an implementation in Scheme [6] and JavaScript [55]. HipHop is not classified in Table 1 because the programming style and evaluation model of synchronous reactive programming languages makes them difficult to compare with the approaches in Table 1. However, there are interesting parallels between some aspects of the HipHop language and the Actor-Reactor Model. HipHop is embedded as a DSL within the Hop language, and Hop code interfaces with HipHop code via *reactive machines* that conceptually fulfil the same role as reactors. Hop code sends input events to a reactive machine and manually triggers a propagation cycle. Output events produced by the HipHop machine can be observed by Hop code via event handlers. Interestingly, one of the core language statements called `atom&` is used within HipHop to execute Hop code, which may contain side-effects and recursive functions. While side-effects in synchronous reactive programs are arguably not subject to the issues discussed in Section 2.2, the authors make a note that the “*execution time [of atom statements] should be kept negligible in practice*” [6, 3.4].

6 Conclusion

To conclude this paper we reflect on our 2 largest contributions, namely identifying the awkward squad for reactive programming and the Actor-Reactor Model. We believe that there is no panacea to write both imperative and reactive programs within a single unified language that exposes the same concepts and operations in both types of programs. Instead, we believe that imperative and reactive programs are fundamentally different, and that they should be programmed whilst guaranteeing their own invariants. To this end the Actor-Reactor Model serves as a new mental model to classify and design reactive systems.

As a secondary contribution, our definition of reactors may prove to be valuable to the field of reactive programming in two ways. First, we define distinct terminology for the different stages of a reactive program: a reactor behaviour represents a dependency graph that is constructed from code, a deployment is a specific instance of a graph that keeps track of all run-time information and state, and a reactor contains the reactive engine that propagates values through one or more deployments. Conceptually, the reactive programs in many existing reactive programming languages are analogous to 1 reactor with 1 deployment. By using our definitions, we open the door for modularity and composition of reactive programs both statically (e.g. via point-wise and point-free composition operators) and dynamically by composing data streams. Second, we conjecture that reactors together with the mechanism of qualification is an alternative, but equally powerful, way to construct higher-order reactive programs. Reactors may provide more insights or clarities with respect to the run-time semantics and resource usage of higher order reactive programs.

The Actor-Reactor Model may inspire designers of reactive languages, framework developers, and researchers to be strict about what can and cannot be programmed within a reactive program, and it may help them to define precisely the rules by which reactive programs interact with their environment. Additionally, the Actor-Reactor Model may be especially useful in application domains where certain parts of a program must be reactive, for example, with specific memory or timing constraints (e.g. robotics and safety systems). Operations that might not fit this model, but which are necessary for program development, are evacuated into actors which are complementary to the reactive program, but which do not violate the invariants of the reactive programming model.

References

- 1 Harold Abelson and Gerald J. Sussman. *Structure and Interpretation of Computer Programs, Second Edition*. MIT Press, 1996.
- 2 José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, Arthur Blot, Benjamin Grégoire, Vincent Laporte, Tiago Oliveira, Hugo Pacheco, Benedikt Schmidt, and Pierre-Yves Strub. Jasmin: High-assurance and high-speed cryptography. In Bhavani M. Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu, editors, *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017*, pages 1807–1823. ACM, 2017. doi:10.1145/3133956.3134078.
- 3 Engineer Bainomugisha, Andoni Lombide Carreton, Tom Van Cutsem, Stijn Mostinckx, and Wolfgang De Meuter. A survey on reactive programming. *ACM Computing Surveys*, 45(4):52:1–52:34, 2013. doi:10.1145/2501654.2501666.
- 4 Gérard Berry. Real time programming: Special purpose or general purpose languages. In Gerhard Ritter, editor, *Information Processing 89, Proceedings of the IFIP 11th World Computer Congress, San Francisco, USA, August 28 - September 1, 1989.*, pages 11–17. North-Holland/IFIP, 1989.


- 5 Gérard Berry and Georges Gonthier. The estereel synchronous programming language: Design, semantics, implementation. *Science of Computer Programming*, 19(2):87–152, 1992. doi:10.1016/0167-6423(92)90005-V.
- 6 Gérard Berry and Manuel Serrano. Hop and hiphop: Multitier web orchestration. In Raja Natarajan, editor, *Distributed Computing and Internet Technology - 10th International Conference, ICDCIT 2014, Bhubaneswar, India, February 6-9, 2014. Proceedings*, volume 8337 of *Lecture Notes in Computer Science*, pages 1–13. Springer, 2014. doi:10.1007/978-3-319-04483-5_1.
- 7 Jonas Bonér, Dave Farley, Roland Kuhn, and Martin Thompson. The reactive manifesto. <https://web.archive.org/web/20191210084324/https://www.reactivemanifesto.org/>. Accessed: 2019-12-10.
- 8 Walter S Brainerd and Lawrence H Landweber. *Theory of computation*. John Wiley & Sons, Inc., 1974. URL: <https://archive.org/details/theoryofcomputat00brai>.
- 9 Andoni Lombide Carreton, Stijn Mostinckx, Tom Van Cutsem, and Wolfgang De Meuter. Loosely-coupled distributed reactive programming in mobile ad hoc networks. In Jan Vitek, editor, *Objects, Models, Components, Patterns, 48th International Conference, TOOLS 2010, Málaga, Spain, June 28 - July 2, 2010. Proceedings*, volume 6141 of *Lecture Notes in Computer Science*, pages 41–60. Springer, 2010. doi:10.1007/978-3-642-13953-6_3.
- 10 Sunjay Cauligi, Gary Soeller, Fraser Brown, Brian Johannesmeyer, Yunlu Huang, Ranjit Jhala, and Deian Stefan. Fact: A flexible, constant-time programming language. In *IEEE Cybersecurity Development, SecDev 2017, Cambridge, MA, USA, September 24-26, 2017*, pages 69–76. IEEE Computer Society, 2017. doi:10.1109/SecDev.2017.24.
- 11 Byron Cook, Andreas Podelski, and Andrey Rybalchenko. Termination proofs for systems code. In Michael I. Schwartzbach and Thomas Ball, editors, *Proceedings of the ACM SIGPLAN 2006 Conference on Programming Language Design and Implementation, Ottawa, Ontario, Canada, June 11-14, 2006*, pages 415–426. ACM, 2006. doi:10.1145/1133981.1134029.
- 12 Gregory H. Cooper and Shriram Krishnamurthi. Embedding dynamic dataflow in a call-by-value language. In Peter Sestoft, editor, *Programming Languages and Systems, 15th European Symposium on Programming, ESOP 2006, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2006, Vienna, Austria, March 27-28, 2006. Proceedings*, volume 3924 of *Lecture Notes in Computer Science*, pages 294–308. Springer, 2006. doi:10.1007/11693024_20.
- 13 Oracle Corporation. JEP 266: More concurrency updates. <https://web.archive.org/web/20191009093608/https://openjdk.java.net/jeps/266>. Accessed: 2019-10-09.
- 14 Evan Czaplicki. A farewell to frp. <https://web.archive.org/web/20191208051242/https://elm-lang.org/news/farewell-to-frp>. Accessed: 2019-12-30.
- 15 Evan Czaplicki and Stephen Chong. Asynchronous functional reactive programming for GUIs. 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 411–422. ACM, 2013. doi:10.1145/2491956.2462161.
- 16 Frank S. de Boer, Vlad Serbanescu, Reiner Hähnle, Ludovic Henrio, Justine Rochas, Crystal Chang Din, Einar Broch Johnsen, Marjan Sirjani, Ehsan Khamespanah, Kiko Fernandez-Reyes, and Albert Mingkun Yang. A survey of active object languages. *ACM Computing Surveys*, 50(5):76:1–76:39, 2017. doi:10.1145/3122848.
- 17 Robert de Simone, Jean-Pierre Talpin, and Dumitru Potop-Butucaru. The synchronous hypothesis and synchronous languages. In Richard Zurawski, editor, *Embedded Systems Handbook*. CRC Press, 2005. doi:10.1201/9781420038163.ch8.
- 18 Joscha Drechsler, Ragnar Mogk, Guido Salvaneschi, and Mira Mezini. Thread-safe reactive programming. *Proceedings of the ACM on Programming Languages*, 2(OOPSLA):107:1–107:30, 2018. doi:10.1145/3276477.

- 19 Jonathan Edwards. Coherent reaction. In Shail Arora and Gary T. Leavens, editors, *Companion to the 24th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2009, October 25-29, 2009, Orlando, Florida, USA*, pages 925–932. ACM, 2009. doi:10.1145/1639950.1640058.
- 20 Jonathan Edwards. Coherent reaction. Technical Report MIT-CSAIL-TR-2009-024, Massachusetts Institute of Technology, Computer Science and Artificial Intelligence Laboratory, Cambridge, 02139 Massachusetts, USA, June 2009. URL: <http://web.archive.org/web/20181103183154/http://dspace.mit.edu/bitstream/handle/1721.1/45563/MIT-CSAIL-TR-2009-024.pdf?sequence=1>.
- 21 Bonnie Eisenman. *Learning React Native: Building Native Mobile Apps with JavaScript*. O’Reilly Media, Inc., 2 edition, 2017.
- 22 Daniel P. Friedman and Mitchell Wand. *Essentials of programming languages (3. ed.)*. MIT Press, 2008.
- 23 Thierry Gautier and Paul Le Guernic. SIGNAL: A declarative language for synchronous programming of real-time systems. In Gilles Kahn, editor, *Functional Programming Languages and Computer Architecture, Portland, Oregon, USA, September 14-16, 1987, Proceedings*, volume 274 of *Lecture Notes in Computer Science*, pages 257–277. Springer, 1987. doi:10.1007/3-540-18317-5_15.
- 24 Adele Goldberg and David Robson. *Smalltalk-80: The Language and Its Implementation*. Addison-Wesley, 1983.
- 25 Nicolas Halbwachs, Fabienne Lagnier, and Christophe Ratel. Programming and verifying real-time systems by means of the synchronous data-flow language LUSTRE. *IEEE Transactions on Software Engineering*, 18(9):785–793, 1992. doi:10.1109/32.159839.
- 26 Daniel Ignatoff, Gregory H. Cooper, and Shriram Krishnamurthi. Crossing state lines: Adapting object-oriented frameworks to functional reactive languages. In Masami Hagiya and Philip Wadler, editors, *Functional and Logic Programming, 8th International Symposium, FLOPS 2006, Fuji-Susono, Japan, April 24-26, 2006, Proceedings*, volume 3945 of *Lecture Notes in Computer Science*, pages 259–276. Springer, 2006. doi:10.1007/11737414_18.
- 27 Facebook Inc. Components and props. <https://web.archive.org/web/20191126131226/http://reactjs.org/docs/components-and-props.html>. Accessed: 2019-11-26.
- 28 Facebook Inc. React: A javascript library for building user interfaces. <https://web.archive.org/web/20191009084855/http://reactjs.org/>. Accessed: 2019-10-09.
- 29 Reactive Streams Initiative. Reactive streams. <https://web.archive.org/web/20191009093755/https://www.reactive-streams.org/>. Accessed: 2019-10-09.
- 30 Joeri De Koster, Tom Van Cutsem, and Wolfgang De Meuter. 43 years of actors: a taxonomy of actor models and their key properties. In Sylvan Clebsch, Travis Desell, Philipp Haller, and Alessandro Ricci, editors, *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. ACM, 2016. doi:10.1145/3001886.3001890.
- 31 Roland Kuhn, Brian Hanafée, and Jamie Allen. *Reactive Design Patterns*. Manning Publications Co., Greenwich, CT, USA, 1st edition, 2017.
- 32 Miran Lipovaca. *Learn You a Haskell for Great Good!: A Beginner’s Guide*. No Starch Press, San Francisco, CA, USA, 1st edition, 2011.
- 33 Ingo Maier and Martin Odersky. Deprecating the observer pattern with scala.react. Technical Report EPFL-REPORT-176887, École Polytechnique Fédérale de Lausanne, EPFL IC IINFCOM LAMP, Station 14, 1015 Lausanne, 2012. URL: <http://web.archive.org/web/20200522141109/https://infoscience.epfl.ch/record/176887>.
- 34 Leo A. Meyerovich, Arjun Guha, Jacob P. Baskin, Gregory H. Cooper, Michael Greenberg, Aleks Bromfield, and Shriram Krishnamurthi. Flapjax: a programming language for ajax applications. In Shail Arora and Gary T. Leavens, editors, *Proceedings of the 24th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2009, October 25-29, 2009, Orlando, Florida, USA*, pages 1–20. ACM, 2009. doi:10.1145/1640089.1640091.

- 35 Phuc C. Nguyen, Thomas Gilray, Sam Tobin-Hochstadt, and David Van Horn. Size-change termination as a contract: dynamically and statically enforcing termination for higher-order programs. In Kathryn S. McKinley and Kathleen Fisher, editors, *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2019, Phoenix, AZ, USA, June 22-26, 2019.*, pages 845–859. ACM, 2019. doi:10.1145/3314221.3314643.
- 36 Bjarno Oeyen, Humberto Rodríguez-Avila, Sam Van den Vonder, and Wolfgang De Meuter. Composable higher-order reactors as the basis for a live reactive programming environment. In Guido Salvaneschi, Wolfgang De Meuter, Patrick Eugster, Lukasz Ziarek, and Francisco Sant’Anna, editors, *Proceedings of the 5th ACM SIGPLAN International Workshop on Reactive and Event-Based Languages and Systems, REBLS@SPLASH 2018, Boston, MA, USA, November 4, 2018*, pages 51–60. ACM, 2018. doi:10.1145/3281278.3281284.
- 37 Simon Peyton Jones. Tackling the awkward squad: monadic input/output, concurrency, exceptions, and foreign-language calls in haskell. *Engineering Theories of Software Construction*, 180:47–96, January 2001. IOS Press.
- 38 ReactiveX. Do. <https://web.archive.org/web/20200415125833/http://reactivex.io/documentation/operators/do.html>. Accessed: 2020-04-15.
- 39 ReactiveX. Introduction. <https://web.archive.org/web/20200415132837/http://reactivex.io/rxjs/manual/overview.html>. Accessed: 2020-04-15.
- 40 ReactiveX. ReactiveX: An api for asynchronous programming with observable streams. <https://web.archive.org/web/20191009085652/http://reactivex.io/>. Accessed: 2019-10-09.
- 41 REScala. REScala manual. <https://web.archive.org/web/20191126124033/http://www.rescala-lang.com/manual/>. Accessed: 2019-11-26.
- 42 REUK. Betz limit. <https://web.archive.org/web/20191025112828/http://www.reuk.co.uk/wordpress/wind/calculation-of-wind-power/>. Accessed: 2019-10-25.
- 43 REUK. Calculation of wind power. <https://web.archive.org/web/20191025113431/http://www.reuk.co.uk/wordpress/wind/betz-limit/>. Accessed: 2019-10-25.
- 44 RexEgg. The explosive quantifier trap. <https://web.archive.org/web/20191223155226/https://www.rexegg.com/regex-explosive-quantifiers.html>. Accessed: 2019-12-23.
- 45 Raymond Roostenburg, Rob Bakker, and Rob Williams. *Akka in action*. Manning Publications Co., 1 edition, 2016.
- 46 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.
- 47 Guido Salvaneschi, Sebastian Proksch, Sven Amann, Sarah Nadi, and Mira Mezini. On the positive effect of reactive programming on software comprehension: An empirical study. *IEEE Transactions on Software Engineering*, 43(12):1125–1143, 2017. doi:10.1109/TSE.2017.2655524.
- 48 Francisco Sant’Anna, Roberto Ierusalimschy, Noemi de La Rocque Rodriguez, Silvana Rossetto, and Adriano Branco. The design and implementation of the synchronous language CÉU. *ACM Trans. Embedded Comput. Syst.*, 16(4):98:1–98:26, 2017. doi:10.1145/3035544.
- 49 RxJS Team. RxJS: Reactive extensions library for javascript. <https://web.archive.org/web/20191125123104/https://rxjs.dev/>. Accessed: 2019-11-25.
- 50 The Flapjax Team. Flapjax framework api documentation. <https://web.archive.org/web/20191128081915/https://www.flapjax-lang.org/docs/>. Accessed: 2019-11-28.
- 51 D. A. Turner. Total functional programming. *Journal of Universal Computer Science*, 10(7):751–768, 2004. doi:10.3217/jucs-010-07-0751.

- 52 GitHub user “htimur”. Akka Streams: Utility function for side effects #23512. <https://web.archive.org/web/20200415164156/https://github.com/akka/akka/issues/23512>. Accessed: 2020-04-15.
- 53 GitHub user “otto-dev”. Request: Overloaded version of .alsoTo that takes a function #28524. <https://web.archive.org/web/20200415154306/https://github.com/akka/akka/issues/28524>. Accessed: 2020-04-15.
- 54 Mandana Vaziri, Olivier Tardieu, Rodric Rabbah, Philippe Suter, and Martin Hirzel. Stream processing with a spreadsheet. In Richard E. 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 360–384. Springer, 2014. doi:10.1007/978-3-662-44202-9_15.
- 55 Colin Vidal, Gérard Berry, and Manuel Serrano. Hiphop.js: a language to orchestrate web applications. In Hisham M. Haddad, Roger L. Wainwright, and Richard Chbeir, editors, *Proceedings of the 33rd Annual ACM Symposium on Applied Computing, SAC 2018, Pau, France, April 09-13, 2018*, pages 2193–2195. ACM, 2018. doi:10.1145/3167132.3167440.
- 56 Zhanyong Wan, Walid Taha, and Paul Hudak. Real-time FRP. In Benjamin C. Pierce, editor, *Proceedings of the Sixth ACM SIGPLAN International Conference on Functional Programming (ICFP '01), Firenze (Florence), Italy, September 3-5, 2001*, pages 146–156. ACM, 2001. doi:10.1145/507635.507654.
- 57 Akinori Yonezawa, Jean-Pierre Briot, and Etsuya Shibayama. Object-oriented concurrent programming in ABCL/1. In Norman K. Meyrowitz, editor, *Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'86), Portland, Oregon, USA, Proceedings.*, pages 258–268. ACM, 1986. doi:10.1145/28697.28722.

A Framework for Resource Dependent EDSLs in a Dependently Typed Language

Jan de Muijnck-Hughes 

University of Glasgow, United Kingdom

Jan.deMuijnck-Hughes@glasgow.ac.uk

Edwin Brady 

University of St Andrews, United Kingdom

ecb10@st-andrews.ac.uk

Wim Vanderbauwhede 

University of Glasgow, United Kingdom

Wim.Vanderbauwhede@glasgow.ac.uk

Abstract

Idris' Effects library demonstrates how to embed resource dependent algebraic effect handlers into a dependently typed host language, providing run-time and compile-time based reasoning on type-level resources. Building upon this work, RESOURCES is a framework for realising Embedded Domain Specific Languages (EDSLs) with type systems that contain domain specific substructural properties. Differing from Effects, RESOURCES allows a language's substructural properties to be encoded within type-level resources that are associated with language variables. Such an association allows for multiple effect instances to be reasoned about autonomously and without explicit type-level declaration. Type-level predicates are used as proof that the language's substructural properties hold. Several exemplar EDSLs are presented that illustrates our framework's operation and how dependent types provide correctness-by-construction guarantees that substructural properties of written programs hold.

2012 ACM Subject Classification Software and its engineering → General programming languages; Software and its engineering → Language features; Software and its engineering → Domain specific languages; Software and its engineering → System modeling languages

Keywords and phrases Dependent Types, Algebraic Effect Handlers, Domain-Specific Languages, Embedded Domain Specific Languages, Idris, Substructural Type-Systems

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2020.20

Category Pearl

Supplementary Material ECOOP 2020 Artifact Evaluation approved artifact available at <https://doi.org/10.4230/DARTS.6.2.2>.

Funding This work was funded by EPSRC projects: *Border Patrol: Improving Smart Device Security through Type-Aware Systems Design* (EP/N028201/1); and *Type-Driven Verification of Communicating Systems* – EP/N024222/1.

Acknowledgements The authors would like to thank the anonymous reviewers for their excellent reviews that served to better the work.

1 Introduction

Substructural Type-Systems allow type-systems to reason about abstract resources associated with the type-system's domain of operation [65]. For general purpose programming languages these resources typically capture, and reason quantitatively about, memory access, variable usage, and erasure of non-essential terms. However, not all languages are general purpose, nor are their abstract resources quantitative in nature cf. Linear Typing with Session



© Jan de Muijnck-Hughes, Edwin Brady, and Wim Vanderbauwhede; licensed under Creative Commons License CC-BY

34th European Conference on Object-Oriented Programming (ECOOP 2020).

Editors: Robert Hirschfeld and Tobias Pape; Article No. 20; pp. 20:1–20:31

Leibniz International Proceedings in Informatics



Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany



Types [30]. Domain Specific Languages (DSLs) are special purpose languages tailored to a specific application domain [26]. Embedded Domain Specific Languages (*EDSLs*) are DSLs that have been embedded within a host language to capitalise upon the host language’s functionality. Implementing an EDSL with a substructural type-system, however, requires an implementation language that not only supports substructural typing but supports reasoning about domain specific substructural properties.

Algebraic effect handlers support reasoning about a program’s side-effects [50] and several programming languages such as OCaml and Haskell have been extended with them [33, 46]. **Effects** [11] is a general purpose *resource dependent* algebraic effect handler library for the dependently typed programming language Idris [10]. Through **Effects**, developers can realise EDSLs with substructural type-systems.

Effects has been realised as *Resource Dependent EDSLs* in which the EDSL is specified as an algebraic data type whose type captures resources that are each associated with an abstract state machine [11]. The EDSL’s type forms a Hoare monad [1, 7] and sequencing of expressions captures valid transitions between individual states of these resources. Such EDSL construction is a common design pattern seen within dependently typed programming languages. For example, there are EDSLs for reasoning about: communicating systems [13, Chp. 15]; communication protocols [19]; and hardware component interfaces [20].

Effects requires, however, that domain specific effects operate within a general purpose effectful context, and effect management is not an autonomic aspect of the program and is the responsibility of the programmer. That is, effect instances describe a single effect within the program, and multiple same effect instances must be explicitly labelled. Figure 1 illustrates these issues with a simple copy function that opens two file handles and writes a single line from one file to another¹. Within the function’s body each same effect instance must be labelled at both the value and type-level. Use of **Effects** is not ideal when designing EDSLs with domain specific effect systems in which multiple same effect instances can occur, nor does the **Effects** library support autonomic effect management.

```
copy : (o, n : String) -> Eff (Maybe FileError) [A ::: FILE (), B ::: FILE (), STDIO]
copy o n = do
  Success <- A :- open o Read | FError e => do {println e; pure (Just e)}
  Result s <- A :- readLine | FError e => do {println e; A :- close; pure (Just e)}
  A :- close
  Success <- B :- open n WriteTruncate | FError e => do {println e; pure (Just e)}
  res <- B :- writeString s
  case res of
    Success => do {B :- close; pure Nothing}
    FError e => do {println e; B :- close; pure (Just e)}
```

■ **Figure 1** Example of labelled effects using Idris’ **Effects** library.

1.1 Contributions

We build on previous work in designing algebraic effect handlers in Idris [11, 10]. Rather than associating an effect’s abstract resource with the program itself we associate it with a bound variable within the EDSL. Further, the list of possible effects is now constrained to an *a priori* set of domain specific effects. Such an association and restriction leads to greater reasoning and manipulation of the effects within a Resource-Dependent EDSLs, thus enabling autonomic effect management and reasoning about the state of an effect’s resource. Given this principal idea, our contributions are:

¹ Idris’ *pattern match & bind* notation reduces the number of case expressions required [11]. This notation supports binding to a value and presentation of the remaining cases on the right.

1. **RESOURCES**, a general purpose framework for constructing Resource Dependent EDSLs that have a domain specific substructural type-systems. Further, we illustrate using *effect handlers* how EDSLs created using **RESOURCES** can be operated on in a variety of different evaluation contexts.
2. A collection of exemplar EDSLs demonstrating the ability of **RESOURCES** to create EDSLs. **Files** reasons about multiple concurrent File IO (Section 4.1); **Wireless** reasons about domain specific bigraph construction (Section 4.2); and **Sessions** captures value dependent global session descriptions – Section 4.3.

RESOURCES is a step forward for developers and presents a new general framework for realising domain specific substructural type-systems for resource dependent EDSLs. Thus, supporting the exploration of novel type-systems similar to those seen in existing systems [47, 28, 17, 32].

1.2 Outline

Section 2 discusses how dependent types support type-level abstract state machines, and reasoning about such machines. Section 3 presents the framework itself, and exemplar EDSLs appear in Section 4. Sections 2 and 3 describes how data types modelled after *De Bruijn indices* [18] provide type-level assertions that certain substructural properties hold.

► **Remark.** Although not essential, before reading about our work we encourage readers not familiar with Idris to learn more about the language, its syntactic constructs, auto-implicit arguments, and semantic highlighting².

2 Type-Level State Tracking and Reasoning

This section introduces the underlying technique for type-level reasoning about abstract resources, through implementation of an EDSL that captures high-level file interactions.

Within our EDSL files are either: closed; open for reading; or open for writing. We encapsulate these operations using the following four operations, and a helper function for displaying showable data: **Open** – which opens a file for reading or writing; **Read** – which reads a string from a file opened for reading; **Write** – which writes a string to a file opened for writing; **Close** – which closes an already open file; and **PrintLn** – which prints showable data.

Parameterised monads allow for language expressions to be associated with a type-level state which we refer to as a *resource* [1]. Hoare monads allow for state transitions to be presented at the type-level [7]. The type of each expression describes how the expressions affects the abstract state. An operation and its type give a Hoare Triple [29]. Definition of state machines within such a monadic construct ensures that any sequence of operations which type checks is a valid sequence of operations. For our example, this means that any operation on a file must respect the type-level state machine we define. Thus, attempting to write to a file opened for reading should present itself as a type error.

Figure 2 presents an implementation of the EDSL within a Hoare monad. **FileIO** is parameterised by the state of the file *before* and *after* each operation. The arguments to **FileIO** are: a **Type**, which represents the return type of the operation; and two **FileState**

² Differing from Idris’ existing colouring scheme, we use a more printer friendly set: **Data** constructors; **Type** constructors; **Bound** variables; named **Function**; Idris **Keywords**; and *Implicitly* bound variables. Agda style highlighting is used for **typed holes**.

20:4 A Framework for Resource Dependent EDSLs in a Dependently Typed Language

instances, which represent the *input* state (the precondition) and the *output* state – the postcondition. These invariants ensure that read and write operations only work when the state of the file is correct: Open for their particular mode of operation. The `PrintLn` operation should not affect the program’s abstract state. The type of `Bind` explains how sequencing changes the file’s state based on a previous expression. `Pure` returns a pure value. Thus, if a sequence of `FileIO` expressions type checks, then it is a valid sequence of operations according to the stated protocol. Rather than use `Bind` and `Pure` directly, do-notation is realised by overloading (`>>=`) and `pure`, with `Bind` and `Pure`.

```
data Mode = R | W

data State = Open Mode | Closed

data FileIO : (type : Type) -> (pre : State) -> (post : State) -> Type where
  Bind : FileIO a stA stB -> a -> FileIO b stB stC -> FileIO b stA stC
  Pure : a -> FileIO a before after

  Open  : (fname : String) -> (m : Mode) -> FileIO () Closed (Open m)
  Read  : FileIO String (Open R) (Open R)
  Write : (value : String) -> FileIO () (Open W) (Open W)
  Close : FileIO () (Open m) Closed
  PrintLn : Show a => a -> FileIO () curr curr
```

■ **Figure 2** An EDSL for interacting with a single file.

Figure 3 presents a sample program written in `FileIO`. The program’s abstract state is initialised to `Closed`. Each expression transitions the state according to the rules embedded in the type of our EDSL. If an incorrect sequence of expressions were to be given, for example opening two files or reading to a file opened for writing, then the program would fail to type-check.

```
toFile : (fname : String) -> (contents : String) -> FileIO () Closed Closed
toFile fname str = do { Open fname W; Write str; Close }
```

■ **Figure 3** An example program for interacting with a single file.

2.1 Files with Errors

The definition for `FileIO` is not sufficiently expressive: Operations on file handles are naturally impure; `FileIO` is pure. The EDSL does not capture potential errors that occur when interacting with a file. For example, being unable to open a file handle, or an error occurring during a read/write operation.

Figure 4a illustrates how the type of `FileIO` can be redefined to address run-time errors. The post-condition is now a function that computes the resulting state dependent on the value returned by the expression. For example, `Open` changes the state to `Closed` in the result of an error, otherwise the state remains the same. The remaining constructors for `FileIOE` can be redefined accordingly. Figure 4b shows Figure 3 rewritten using `FileIOE`. If the result of `Open` or `Write` is not checked, the subsequent interactions will not type check. The next state would be unknown.


```

data FileIOE : (type : Type) -> (pre : State) -> (post : type -> State) -> Type where
  Open : (fname : String)
        -> (m : Mode)
        -> FileIOE (Maybe FileError)
            Closed
            (\res => case res of {Nothing => Open m; Just err => Closed})
  ...

```

(a) Partial redefinition of FileIO.

```

toFile : String -> String -> FileIOE (Maybe FileError) Closed (const Closed)
toFile fname str = do
  Nothing <- Open fname W          | Just err => do {PrintLn err; pure err}
  Nothing <- Write fh "A string" | Just err => do {PrintLn err; Close fh; pure err}
  Close fh
  pure Nothing

```

(b) Figure 3 rewritten as an FileIOE instance.

■ **Figure 4** Redefining FileIO to include type-level enforcement of error handling.

This pattern of state-aware EDSL construction allows reasoning about the abstract state of an EDSL at *compile-time*, based on data obtained at *run-time*. However, FileIOE is not expressive enough to reason about, nor interact with, multiple files.

2.2 Modelling Multiple File Access with Errors

Figure 5 extends the definition of FilesIOE with a list of abstract state machines: one per open file.

```

data FilesIOE : (ty : Type) -> (old : List Item) -> (new : ty -> List Item) -> Type where
  Pure : (val : a) -> FilesIOE a (st val) st

  Bind : FilesIOE a first snd_fn
        -> ((x : a) -> FilesIOE b (snd_fn x) third_fn)
        -> FilesIOE b first third_fn

  Open : (fname : String) -> (m : FMode)
        -> FilesIOE (Either FileError Handle) old
            (\res => case res of {Right hdl => MkItem hdl (Open m)::old; Left _ => old})

  Read : (hdl : Handle) -> (prf : Any (IsOpenFor hdl R) item old)
        -> FilesIOE (Either FileHandle String) old
            (\res => case res of {Right _ => old; Left _ => update (closeHandle) old prf})

  Write : (hdl : Handle) -> (str : String) -> (prf : Any (IsOpenFor hdl W) item old)
        -> FilesIOE (Maybe FileError) old
            (\res => case res of {Nothing => old; Just _ => update (closeHandle) old prf})

  Close : (hdl : Handle) -> (prf : Any (IsHandle hdl) item old)
        -> FilesIOE () old (const $ drop old prf)

  PrintLn : Show a => (msg : a) -> FilesIOE () old (const old)

```

■ **Figure 5** An EDSL to model multiple concurrent file interactions.

To help with reasoning about multiple files we introduce two helper data structures.

```

data Handle = MkHandle
data Item = MkItem Handle FileState

```

Handle represents file handles at both the value and type level, and Item associates a type-level file state with a particular handle. File handles are bound to names using the Bind constructor. Although, we could use a nameless representation based on *De Bruijn* indices

we can take advantage of Idris' elaborator to distinguish between different instances of `Handle` based on their bounded names and type-level values. During the type checking process Idris' elaborator translates high level Idris code to the internal type theory representation [9] by expanding high level language constructs such as case blocks and where clauses, and inferring values for implicit arguments by unification and search. Further, by replacing the original state that parameterises `FileIOE` with a list of these `Item` instances, the state of each open file handle in our EDSL can be tracked.

Figure 6 presents two type-level *predicates* for reasoning about individual items in our type-level context. `IsOpenFor` declares that the given file handle has been opened for reading or writing; and `IsHandle` declares that the given file handle exists. To aid reasoning about multiple file handles, i.e. any item in the program's context, the list quantifier `Any` is used. The `Any` represents existential quantification that the given predicate holds on a list item.

With Idris' do-notation let-bindings are provided, however, let-bindings do not interact with the type-level context. We can use the `Any` list quantifier in conjunction with Idris' elaborator to ensure that aliased variables cannot be used. If an operation with an aliased handle were to be used then proof (witness) cannot be given of the handle's existence in the type-level context as Idris' elaborator will fail to associate the aliased named with an abstract state.

```
data IsOpenFor : (hdl : Handle) -> (mode : FMode) -> (item : Item) -> Type where
  FileIsOpenFor : (m : FMode) -> IsOpenFor hdl m (MkItem hdl (Open m))
```

(a) Predicate for reasoning about file handle mode.

```
data IsHandle : (hdl : Handle) -> (item : Item) -> Type where
  FileExists : (hdl : Handle) -> IsHandle hdl (MkItem hdl st)
```

(b) Predicates for linking file handle to instance of `Handle`.

■ **Figure 6** Predicates.

Figure 7 presents the type signatures for two helper functions that manipulate the type-level context based on an expression's associated predicates. The first function, `update`, updates specific elements in our context dependent upon the supplied `Any` proof. Further, the update function `f` facilitates access to the predicate that holds over the item we are updating. The `drop` function removes an item from the context using the supplied `Any` proof about the item.

```
update : (f : (i : Item)
          -> (prf : p i)
          -> Item)
        -> (context : List Item)
        -> (index : Any p item context)
        -> List Item
drop : (context : List Item)
      -> (index : Any p item context)
      -> List Item
```

(a) Updating an `Item` instance.

(b) Removing an `Item` instance.

■ **Figure 7** Functions for manipulating the type-level context.

With these extra data structures, and predicates, type-level operations on individual file handles in `FilesIOE` becomes autonomic. Files are opened using `Open` which extends the context with the new file handle, its initial state. The old context is retained if the operation fails. Reading a file, using `Read`, requires proof that the file is already open for reading. We do so using `IsOpenFor` and `Any`. If the read is successful then the old context is retained. If the read is unsuccessful then the state of the file is updated to `IsClosed`. The definition of `Write` is analogous to `Read`. Closing a file (`Close`) removes the file's state from the context. For a close operation to be allowed, evidence must be presented that the file *is* in the closed state. With this evidence the file's associated state can be removed.

Figure 8a presents an example of a program written using `FilesIOE`. For each language expression that requires a predicate a proof must also be given. With this approach value level expressions become incredibly verbose. This is *too* verbose. Users should not be expected to write such proofs by hand. Idris supports *auto-implicit* arguments, in which the values for implicit arguments to a function can be automatically constructed using a greedy constructor-based search to find a value that matches the arguments type. By wrapping each language expression in a function that uses auto implicits we can automatically construct the proofs.

```
copy : (old, new : String) -> FilesIOE (Maybe FileError) Nil (const Nil)
copy old new = do
  Right fh <- Open old R | Left err => do {PrintLn err; Pure (Just err)}
  Right str <- Read fh (H $ FileIsOpenFor R) | Left err => do
    PrintLn err
    Close fh (H $ FileExists fh)
    Pure (Just err)
  Close fh (H $ FileExists fh)
  Right fh1 <- Open new W | Left err => do {PrintLn err; Pure (Just err)}
  res <- Write fh1 str (H $ FileIsOpenFor W)
  case res of
    Nothing => do {Close fh1 (H $ FileExists fh); Pure (Nothing)}
    Just err => do {PrintLn err; Close fh1 (H $ FileExists fh1); Pure (Just err)}
```

(a) With Proofs.

```
copy : (old, new : String) -> FilesIOE (Maybe FileError) Nil (const Nil)
copy old new = do
  Right fh <- open old R | Left err => do {println err; pure (Just err)}
  Right str <- read fh | Left err => do
    println err
    close fh
    pure (Just err)
  close fh
  Right fh1 <- open new W | Left err => do {println err; pure (Just err)}
  res <- write fh1 str
  case res of
    Nothing => do {close fh1; pure (Nothing)}
    Just err => do {println err; close fh1; pure (Just err)}
```

(b) With Proofs calculated using auto-implicit arguments.

■ **Figure 8** Figure 3 rewritten using `FilesIOE`.

For example, the wrapper function for `Close` would be written as:

```
close : (h : Handle) -> {auto idx : Any (IsHandle h) i o}
  -> FileIOE () o (const (drop o idx))
close h {idx} = Close h idx
```

By convention, the function that calculates an auto-implicit argument is named using lower case variants of the constructor name. Figure 8b presents the “cleaned” version of Figure 8a.

Notice that for each branch in our case-splits, and bind operations, we must close open file handles. Here the type-level state requires us to exit functions with an empty context. This ensures that all file handles are closed when we exit our program. To ensure that our programs start and end with the correct states the type-synonym `FilesIOE` is defined to ensure that the end state of the program must be empty, implying that all file handles that were open, were also closed.

```
FileIO : Type -> Type
FileIO ty = FilesIOE ty Nil (const Nil)
```

3 The Framework

The definition of `FilesIOE` follows a pattern of EDSL construction seen in existing work [13, 19, 20]. This section describes the implementation of `RESOURCES` that encapsulates the common structures and definition common to these EDSLs.

3.1 Capturing Abstract State

Central to the framework's operation is associating variables with an abstract state that is reasoned about at the type-level. Figure 9 presents the definitions for variables, their associated state, and an EDSLs' context. The relationship between a variable and a state is captured by indexing the type for variables (`Var`) and state items (`StateItem`) with a data type that acts as a meta-type representing the type of the variable's associated state. Following from the *Well-Typed Interpreter* [3], the type of `StateItem` is further indexed by a function to compute the concrete type associated with the type-level value. The definition of `StateItem` associates an instance of `Var` with a specific instance of state. As we saw in Section 2.2, Idris' elaborator allows us to distinguish between different instances of `Var`. The list of state items captured at the type level, the EDSLs context, is collected in a bespoke data type `Context`.

```

data Var : (Ty : Type) -> (ty : Ty) -> Type where
  MkVar : Var type value

data StateItem : (ty : Type) -> (calcSTy : ty -> Type) -> (value : ty) -> Type where
  MkStateItem : (value : type)
    -> (label : Var type value)
    -> (state : calcSTy value)
    -> StateItem type calcSTy value

data Context : (type : Type) -> (calcSTy : type -> Type) -> Type where
  Nil : Context type calcSTy
  (::) : (item : StateItem type calcSTy value)
    -> (rest : Context type calcSTy)
    -> Context type calcSTy

```

■ Figure 9 Definitions for variables, state items, and type-level context.

3.2 Sequencing Language Expressions

Figure 10 presents the parameterised data type that captures state transitions between different abstract states. A type-synonym ensures that all languages defined using the framework use the same signature. A language expression has an expression type (`exprTy`), an existing `Context` instance `pre`, and a function `postK` to compute the new context from the expression's value. `Lang` is a function that constructs an instance of this type signature with the meta-type and a function to compute concrete states indexing the signature.

```

Lang : (type : Type) -> (type -> Type) -> Type
Lang type calcSTy = (exprTy : Type)
  -> (pre : Context type calcSTy)
  -> (postK : exprTy -> Context type calcSTy)
  -> Type

```

■ Figure 10 The type for all EDSLs.

Figure 11 presents `LANG`, a single data type, to collate the: meta-type – (`type`); interpreter (`calcSTy`); and `Lang` instance together.

```

data LANG : Type -> (type : Type) -> (calcSTy : type -> Type) -> Type where
  MkLang : (type : Type)
    -> (calcSTy : type -> Type)
    -> Lang type calcSTy
    -> LANG type calcSTy

```

■ **Figure 11** Data Structure and accessors to hold EDSL Specifications.

Figure 12 presents `LangM`, the data structure that captures, generically, the sequencing of EDSL language expressions. The data type `LangM` removes the need for each EDSL to provide the same definitions for sequencing expressions. The constructor `Value` returns a pure value. `Let` provides sequencing of expressions and insertion of computed values into subsequent expressions. `Expr` provides embedding of EDSL language expressions into `LangM`. The type of `LangM` is indexed by: `m` – a monadic context; `exprTy` – the type associated with an expression; `spec` – the language specification that is being sequenced; `pre` – the original context; and `postK` – the computed context.

```

data LangM : (m : Type -> Type)
  -> (exprTy : Type)
  -> (spec : LANG type calcSTy)
  -> (pre : Context type calcSTy)
  -> (postK : exprTy -> Context type calcSTy)
  -> Type where
  Value : (value : a) -> LangM m a spec (postK value) postK

  Let : LangM m a spec old oldK
    -> ((val : a) -> LangM m b spec (oldK val) postK)
    -> LangM m b spec old postK

  Expr : {eSig : Lang type calcSTy}
    -> (expr : eSig a pre postK)
    -> LangM m a (MkLang type calcSTy eSig) pre postK

```

■ **Figure 12** Definition of `LangM`.

3.3 Reasoning About Abstract State

Within dependently typed languages, list quantifiers such as `All` and `Any` are based on *De Bruijn* indices and reasoning about all or specific elements within a standard list using a provided predicate [18]. Figure 13 presents similar predicated quantifiers that can be constructed for `Context`. The `AllContext` predicate mirrors `All` and allows one to present a predicate that applies to all state items. Mirroring `Any`, `InContext` constructs a proof that there is an element (searching from the head of the list) satisfying the provided predicate.

```

data AllContext : (p : (value : type) -> (item : StateItem type calcSTy value) -> Type)
  -> (c : Context type calcSTy)
  -> Type where

data InContext : (value : type)
  -> (p : StateItem type calcSTy value -> Type)
  -> (c : Context type calcSTy)
  -> Type where

```

■ **Figure 13** Quantifiers for reasoning about elements in `Context`.

Generic functions can be constructed using these quantifiers to operate on `Context` instances. Specifically, instances of `InContext` provide type-safe transformations on specific elements. Figure 14 presents the definition of several of these functions. The first function, `update`, updates specific elements in our context dependent upon the supplied `InContext` proof. Here, the update function `f` facilitates access to the predicate that holds over the item being updated. The function `drop` removes an item from the context using the `InContext` proof about the item. A third function `setState` allows the state to be replaced.

```

update : (context : Context type calcSTy)
  -> (index   : InContext value predicate context)
  -> (f : (item : StateItem type calcSTy value)
    -> (prf  : predicate item)
    -> StateItem type calcSTy value)
  -> Context type calcSTy

drop : {predicate : StateItem type calcSTy value -> Type}
  -> (context   : Context type calcSTy)
  -> (index     : InContext value predicate context)
  -> Context type calcSTy

setState : {predicate : StateItem type calcSTy value -> Type}
  -> (context   : Context type calcSTy)
  -> (index     : InContext value predicate context)
  -> (item'    : calcSTy value)
  -> Context type calcSTy

```

■ **Figure 14** Functions acting on `Context` instances.

3.4 Language Evaluation

The `Effects` library uses `Idris` interfaces to link effect specifications (descriptions) to implementation handlers that realise the specification for a specific implementation context. This is the `Handler` interface. Figure 15 presents a similarly named interface to describe EDSL evaluation and effect handling, together with a secondary interface, `RealVar` that details how variables in an EDSL are to be translated to concrete types. Within `RESOURCES` our individual effect specifications will be subterms in our EDSL and their handlers the corresponding body in the implementation.

The `Handler` interface is indexed by: the meta-type type; the meta-type interpreter; a language expression specification; an accumulator; and a specific evaluation context. Similarly to the `Effects` handler interface, instances of `Handler` detail how to evaluate EDSL expressions in a specific evaluation context, and how the domain specific effects are to be handled. The function `handle` takes an evaluation environment, the expression to be considered, an accumulator, and a continuation to pass on the updated environment and accumulator.

Figure 16 presents the definition (`Env`) for evaluation environments to keep track of variables and their abstract state. The type of `Env` is indexed by an evaluation context `m` and the current state of the EDSL (`ctxt`) during evaluation. This ensures that the items in the environment grows and shrinks as the type-level context (`ctxt`) grows and shrinks. The data type `Tag` is a container for holding concrete variable representations. The function of the `RealVar` interface computes the concrete type from the language’s meta-type.

Section 3.3 presented predicates for reasoning about state items in instances of `Context`. These same predicates are used to provide operations on our computation environments; `Env` is indexed by a context. Figure 17 presents the function definitions for `lookup`, `update`, and `drop` that mirror the functions presented in Section 3.3. When specifying how EDSLs are evaluated, type-level operations on the context *must* be mirrored at the value level for the environment.

```

interface RealVar (type : Type) where
  CalcRealType : type -> Type

interface RealVar type
  => Handler (type : Type) (eSig : Lang type) (calc : type -> Type)
        (tyAcc : Type) (m : Type -> Type) | type
  where
    handle : (env : Env m type pre)
      -> (expr : eSig tyExpr pre postK)
      -> (acc : tyAcc)
      -> (cont : (value : tyExpr)
          -> (env' : Env m type (postK value))
          -> (acc' : tyAcc)
          -> m tyRes)
      -> m tyRes

```

■ **Figure 15** Interfaces for evaluation.

```

data Tag : (type : Type) -> (value : type) -> Type where
  MkTag : RealVar type => (real : CalcRealType value) -> Tag type value

data Env : (m : Type -> Type) -> (ty : Type) -> (ctxt : Context ty calcSTy) -> Type where
  Nil : Env m type Nil
  (::) : RealVar type
    => {item : StateItem type calcSTy value}
    -> (tag : Tag type value)
    -> (rest : Env m type items)
    -> Env m type (item::items)

```

■ **Figure 16** Evaluation environment definition.

```

lookup : RealVar ty
  => (env : Env m ty ctxt)
  -> {p : (item : StateItem ty calcSTy value) -> Type}
  -> (idx : InContext value p ctxt)
  -> Tag ty value

(a) Lookup items from environment.
update : RealVar ty
  => (env : Env m ty ctxt)
  -> (idx : InContext value p ctxt)
  -> (up : (i : StateItem ty calcSTy value)
      -> p i
      -> StateItem ty calcSTy value)
  -> Env m ty (update ctxt)

drop : (env : Env m ty ctxt)
  -> (idx : InContext value p ctxt)
  -> Env m ty (drop ctxt idx)

(c) Remove items from environment.

(b) Update items from environment.

```

■ **Figure 17** Functions operating over an execution environment.

Figure 18 presents the generic function `run` that evaluates languages defined in `RESOURCES`. As arguments the function `run` takes: a closed `LangM` program (`prog`); and an initial seed for the accumulator – `init`. On successful evaluation the function returns the result of evaluating `prog` and the final state of the accumulator. The type of the function has been further constrained with `Applicative` to return the result of the evaluation within the context of the environment `m`. For pure evaluation contexts, i.e. identity, a separate `runPure` function can be defined that need not be constrained by `Applicative`.

```

run : (Applicative m, Handler type lang tyAcc m)
    => (init : tyAcc)
    -> (prog : LangM m tyExpr lang c Nil (const Nil))
    -> m (Pair tyExpr tyAcc)

```

■ **Figure 18** Run function.

4 Exemplar Uses of Resources

This section explores use of RESOURCES through the construction of three separate EDSLs. For each EDSL presented we only present salient aspects of the construction. The complete definitions are available in the accompanying artefact.

The first, **Files**, replicates the running example from Section 2 demonstrating how to build the EDSL and specify a handler for the **IO** computation context. The second, **Wireless** presents a EDSL for describing wireless connections between mobile devices, and details a **Handler** instance for constructing a *BiGraph* representation. The last EDSL, **Sessions** replicates salient aspects from, and extends the functionality, of **Sessions** an EDSL for describing communication protocols [19], and shows a simpler construction using RESOURCES.

4.1 Exemplar 1: Reasoning About Multiple File Handles

This section demonstrates how to use RESOURCES to re-implement the **FilesIOE** EDSL from Section 2.1.

4.1.1 EDSL Definition

Figure 19 presents the type-level definitions required by the EDSL. Like **FilesIOE**, there is a *single* state machine captured within the EDSL's type. The type **FH** is a singleton type acting as a meta-type for the state machine, and the type synonym **FileHandle** acts a convenient wrapper when referring to file handles. **FHStateType** is the function that calculates the state type based on **FH**, and **FileStateItem** is the type synonym for representing the EDSLs abstract states. While this construction is cumbersome for single state-machine EDSLs, Section 4.2 demonstrates how this construction can support multiple type-level state machines.

<code>data FH = MkFH</code>	<code>FHStateType : FH -> Type</code>
(a) EDSL Metatype.	<code>FHStateType _ = FileState</code>
<code>FileHandle : Type</code>	(c) Interpreter to compute state type.
<code>FileHandle = Var FH MkFH</code>	<code>FileStateItem : Type</code>
(b) Alias to represent file handles.	<code>FileStateItem = StateItem FH FHStateType MkFH</code>
	(d) Type synonym to represent state items.

■ **Figure 19** Preliminary definitions and example predicate.

Figure 20 presents the algebraic data type (**Files**) that captures the language's expressions. Notice how the definitions mirror that of **FilesIOE** from Section 2.1. Rather than use explicit case statements in anonymous functions, named functions are provided that compute the state transitions. As an example we present the function definition for **readTrans**:

```

readTrans : Either FileError String
    -> (old : Context FH FHStateType)

```



```

data Files : Lang FH FHStateType where
  Open : (fname : String)
        -> (fm : FMode)
        -> Files (Either FileError (FileHandle) old (\res => openTrans res fm old)

  Read : (hdl : FileHandle)
        -> (prf : InContext MkFH (IsOpenFor hdl R) old)
        -> Files (Either FileError String) old (\res => readTrans res old prf)

  Write : (hdl : FileHandle)
         -> (msg : String)
         -> (prf : InContext MkFH (IsOpenFor hdl W) old)
         -> Files (Maybe FileError) old (\res => writeTrans res old prf)

  Close : (hdl : FileHandle)
         -> (prf : InContext MkFH (IsHandle hdl) old)
         -> Files () old (const $ drop old prf)

  PrintLn : Show a => a -> Files () old (const old)

```

■ **Figure 20** Definition for `Files`.

```

-> InContext MkFH (IsOpenFor hdl R) old
-> Context FH FHStateType
readTrans (Right _) old _ = old
readTrans (Left _) old prf = update old prf (\i,p => closeHandle i p)

```

`Files` uses two predicates to reason about a file handle's abstract state: `IsOpenFor` and `IsHandle`. Their definition mirrors that to those provided in Figure 6. As an example, we present only the new definition for `IsOpenFor`:

```

data IsOpenFor : FileHandle -> FMode -> FileStateItem -> Type where
  FileIsOpenFor : (m : FMode)
                 -> IsOpenFor hdl m (MkStateItem MkFH hdl (Open m))

```

The language definition for `Files` is thus:

```

FILES : LANG FH FHStateType
FILES = MkLang FH FHStateType Files

```

The generic computation context `LangM` uses `LANG` instances to ensure correct embedding of EDSL expressions. Expressions can then be embedded within `LangM` using `expr` as follows:

```

openFile : (fname : String) -> (fm : FMode)
         -> LangM m (Either FileError (FileHandle)) FILES old
         (\res => openTrans res fm old)
openFile fname fm = expr $ Open fname fm

```

4.1.2 Handler for the `Files` EDSL

Figure 21 presents the handler definition for the `IO` computation context. An implementation of `RealVar` maps the singleton type `FH` to a *real* file handle. The accumulator has the unit type as this implementation of `Handler` only evaluates `File` expressions. The accumulator is not required. Each of the expression handlers realises the requisite file operations, and follows that of the `FILE` effect [10]. Within our implementation, however, our environment (`env`) keeps track of the open file handles.

```

RealVar FH where
  CalcRealType MkFH = File

Handler FH FHStateType Files () IO where
  handle env (Open fname fm) acc cont = do
    let m = case fm of {R => Read; W => WriteTruncate}
        res <- openFile fname m
        case res of
          Left err => cont (Left err) env acc
          Right fh => cont (Right MkVar) (MkTag fh::env) acc

  handle env (Read hdl prf) acc cont = do
    let MkTag fh = lookup env prf
        res <- fGetLine fh
        case res of
          Left err => cont (Left err) (update env prf (\i,p => closeHandle i p)) acc
          Right str => cont (Right str) env acc

  handle env (Write hdl str prf) acc cont = do
    let MkTag fh = lookup env prf
        res <- fPutStrLn fh str
        case res of
          Left err => cont (Just err) (update env prf (\i,p => closeHandle i p)) acc
          Right _ => cont Nothing env acc

  handle env (Close hdl prf) acc cont = do
    let MkTag fh = lookup env prf
        closeFile fh
        cont () (drop env prf) acc

  handle env (PrintLn a) acc cont = do
    printLn a
    cont () env acc

```

■ Figure 21 Handler instance for Files.

4.1.3 Example Programs

Figure 22 presents two example programs written using Files. The first (Figure 22a) replicates the running example presented in Figures 1 and 8. The second example (Figure 22b) demonstrates an incomplete program, indicated by the typed-hole, that will fail to type check. This is because the file has been opened for reading and we are attempting to write to the file. The typed hole is required in this example to ensure that the example can begin to type-check. RESOURCES ensures that the substructural checks are performed at compile time.

4.2 Exemplar 2: Constructing Domain Specific Bigraphs

This next example examines bigraphs, a mathematical model for representing the communication made between entities and said entities physical placement [40]. A bigraph comprises of a *place graph* that denotes the spatial relations between entities, and a *link graph* that denotes the communication relations. Each entity within a bigraph is typed with a domain specific construct that dictates the entity's: *arity* – number of links; and *atomicity* – containment of other entities. Existing bigraph constructions make their bigraphs abstract (entities are identifier-free) and refer to entities using singleton types [53].

Figure 23 presents a commonly used algebraic notation for bigraph specification. The standard algebraic bigraph definition embeds the link graph within an entities definition in which the type of the entity dictates the arity. The number of links K possesses is determined

```

copy : (a,b : String) -> Files m (Maybe FileError)
copy a b = do
  Right fh <- openFile a R | Left err => do {println err; pure (Just err)}
  Right s <- readString fh | Left err => do {println err; closeFile fh; pure (Just err)}
  closeFile fh
  Right fh1 <- openFile b W | Left err => do {println err; pure (Just err)}
  res <- writeString fh1 s
  case res of
    Nothing => do {closeFile fh1; pure Nothing}
    Just err => do {println err; closeFile fh1; pure (Just err)}

```

(a) Example from Figures 1 and 8.

```

copy : (a,b : String) -> Files m (Maybe FileError)
copy a b = do
  Right fh <- openFile a R | Left err => do {println err; pure (Just err)}
  Right s <- readString fh | Left err => do {println err; closeFile fh; pure (Just err)}

  writeString fh s
  ?remainder

```

(b) A failing example.

■ **Figure 22** Example instances of `Files`.

by its arity. Bigraphs can be *nested*, situated beside each other using a *merge product*, or associated together using *parallel product*. The internal structure of a bigraph entity can be abstracted away using `id`. Closure of names allows one to define internal links between entities, and free names represent external connections. Bigraphs also enjoy an expressive graphical notation which we do not detail here.

$P \cdot Q$	Nesting	(1)	<code>id</code>	Identity	(4)
$P Q$	Merge product	(2)	$K_{x,y}$	An entity of type K with names x, y	(5)
$P Q$	Parallel product	(3)	$/x P$	Closure of name x in P	(6)

■ **Figure 23** Algebraic Definition for Bigraphs.

The algebraic structure of bigraphs are general purpose and restrictions on the bigraph's shape is guided by a *system of sorts*. These sorts presents a series of side conditions on the link and place graph. Application of this system is often left as an aside from the bigraph itself. Using `RESOURCES` we can show how to build an EDSL that encapsulates the system of sorts and when interpreted produces a bigraph instance.

4.2.1 Domain Model

Existing work has introduce a bigraph model for representing Wireless Sensor Networks (WSNs) [54]. In their model they use the place graph of bigraphs to model the physical deployment of nodes, together with their configuration, and applications running on said nodes. The link graph connects data, applications, and nodes together. In this example we take a reduced version of their system of sorts to describe sending of messages between mobile devices and laptops that are connected over a wireless network. For simplicity, we restrict number of concurrent connects laptops have to ten, and mobile devices to two. Devices are located in rooms that are within buildings.

Table 1 presents the description of our example’s types and sorts. Buildings can only contain rooms and cannot be linked over for communication. Similarly, rooms can only contain devices. Devices contain only messages, and laptops and mobiles have an arity respective to their max number of connections. Messages cannot contain other entities.

■ **Table 1** Types and Sorts for representing entities in *Wireless*.

Entity	Arity	Usage Restrictions
Building	0	Complex for Rooms only.
Room	0	Complex for Devices only.
Device Laptop	10	Complex for Messages only.
Device Mobile	2	Complex for Messages only.
Messages	0	Atomic

Figure 24 presents an example bigraph instance using the system from Table 1. We situate two buildings that contain potentially many rooms next to each other, and describe some rooms within them. Within one room in the first building, a laptop is situated that is connected to another laptop in the other building, together with a mobile device (with a message) that is connected to a laptop in an adjacent room in the same building.

$$\begin{aligned} & /m/n (\text{Building} \cdot (\text{Room} \cdot (\text{Laptop}_{\{n\}} \mid (\text{Mobile}_{\{m\}} \cdot \text{Message})) \mid (\text{Room} \cdot (\text{Laptop}_{\{m\}}))) \mid \text{id}) \\ & \parallel ((\text{Building} \cdot (\text{Room} \cdot \text{Laptop}_{\{n\}} \mid \text{id}))) \end{aligned}$$

■ **Figure 24** Example Bigraph instance using algebraic notation.

4.2.2 EDSL Definition

Figure 25a presents a realisation for Table 1 using standard Idris constructs. Types are presented as an enumerated type, in which we coalesce the definition for devices. For rooms and messages we keep track of their allocation into entities, and for devices we keep track of their allocation and number of free connections. Buildings do not have an associated abstract state. The function `maxConn` calculates a devices arity, this function is used in secondary function `defState` (not defined) that constructs `Stated` instances.

```

data DTy = MOBILE | LAPTOP
data Ty = ROOM | BLDG | MSG
         | DEVICE DTy
(a) Metatypes.

maxConn : DTy -> Nat
maxConn MOBILE = 2
maxConn LAPTOP = 10
(b) Function to compute device arity.

data Stated : DTy -> Type where
  MkD : Bool -> Nat -> Stated ty
(c) State for devices.

CalcStateType : Ty -> Type
CalcStateType ROOM = Bool
CalcStateType BLDG = ()
CalcStateType (DEVICE ty) = Stated ty
CalcStateType MSG = Bool
(d) Function to compute state types.

```

■ **Figure 25** Preliminary definitions.

Figure 26 presents the language definition for *Wireless*. Introduction of entities extend the abstract state: buildings have no state; rooms and messages are initially unassigned; and devices are initialised not allocated and connection free.

```

data Wireless : Lang Ty CalcStateType where
  NewBuilding : Wireless (Var Ty BLDG) old (\lbl => MkStateItem BLDG lbl () :: old)

  NewRoom : Wireless (Var Ty ROOM) old (\lbl => MkStateItem ROOM lbl False :: old)

  NewDevice : (type : DTy)
    -> Wireless (Var Ty (DEVICE type)) old
      (\lbl => MkStateItem (DEVICE type) lbl (defState type) :: old)

  NewMessage : Wireless (Var Ty MSG) old (\lbl => MkStateItem MSG lbl False :: old)

  Insert : (varX : Var Ty x)
    -> (varY : Var Ty y)
    -> (prfValid : ValidAssign y x)
    -> (prfFree : InContext x (Unassigned x varX) old)
    -> (prfInsert : InContext y (CanAssign y varY) (update old prfFree Use))
    -> Wireless () old (const $ update (update old prfFree Use)
      prfInsert (Assign varX prfValid))

  Link : (varX : Var Ty (DEVICE typeX))
    -> (varY : Var Ty (DEVICE typeY))
    -> (prfSpaceX : InContext (DEVICE typeX) (CanConnect varX) old)
    -> (prfSpaceY : InContext (DEVICE typeY)
      (CanConnect varY)
      (update old prfSpaceX Connect))
    -> Wireless () old
      (const $ update (update old prfSpaceX Connect) prfSpaceY Connect)

  End : Wireless () old (const Nil)

```

■ **Figure 26** Definition for `Wireless`.

The constructor `Insert` is a generic expression that supports: insertion of rooms into buildings; devices into rooms; and messages into devices. For insertion of entity `varX` into `varY` to take place several checks are performed. First we check to see if the entities of type `x` and `y` are valid assertions using `ValidAssign` defined in Figure 27b. We then check to see if the child entity (`varX`) has already been inserted. The predicate `Unassigned` (Figure 28a) attests to this, and the function `Use` (Figure 28b) updates the context accordingly. The final check is to see if the parent entity (`varY`) can be assigned to. By design, the predicate `CanAssign` (Figure 27a) uses type level pattern matching to reason about abstract states that can contain other entities, and for a device that there is at least one free connection left. The function `Assign` (Figure 27c) updates the context accordingly.

```

data CanAssign : (thisValue : Ty)
  -> (thisVar : Var Ty valueThis)
  -> (item : StateItem Ty CalcStateType valueThis)
  -> Type where
  ToABuilding : CanAssign BLDG bld (MkStateItem BLDG bld ())
  ToARoom : CanAssign ROOM rm (MkStateItem ROOM rm True)
  ToADevice : CanAssign (DEVICE ty) dev (MkStateItem (DEVICE ty) dev (MkD True (S n)))

```

(a) Predicate.

```

data ValidAssign : Ty -> Ty -> Type where
  ValidBR : ValidAssign BLDG ROOM
  ValidRD : ValidAssign ROOM (DEVICE ty)
  ValidDM : ValidAssign (DEVICE ty) MSG
  Assign : Var Ty x
    -> ValidAssign v x
    -> (i : StateItem Ty CalcStateType v)
    -> CanAssign value lbl i
    -> StateItem Ty CalcStateType v

```

(b) Side-Condition.

(c) Update Function.

■ **Figure 27** Predicates and update function for reasoning about association of nodes.

```

data Unassigned : (value : Ty) -> (lbl : Var Ty value)
  -> (item : StateItem Ty CalcStateType value) -> Type where
  URoom      : Unassigned ROOM      rm (MkStateItem ROOM      rm False)
  UDevice    : Unassigned (DEVICE ty) dev (MkStateItem (DEVICE ty) dev (MkD False c))
  UMessage   : Unassigned MSG       msg (MkStateItem MSG       msg False)

```

(a) Predicate.

```

Use : (item : StateItem Ty CalcStateType value)
  -> (prf : Unassigned value lbl item)
  -> StateItem Ty CalcStateType value

```

(b) Update Function.

■ **Figure 28** Predicate and Function for Assigning Variables.

```

data CanConnect : (to : Var Ty (DEVICE type))
  -> (item : StateItem Ty CalcStateType (DEVICE type))
  -> Type where
  HasSpace : CanConnect dev (MkStateItem (DEVICE type) lbl (MkD True (S n)))

```

(a) Predicate.

```

Connect : (item : StateItem Ty CalcStateType (DEVICE type))
  -> (prf : CanConnect to item)
  -> StateItem Ty CalcStateType (DEVICE type)

```

(b) Update Function.

■ **Figure 29** Predicates and update function for reasoning about connection of devices.

Notice for `Insert` we have had to update the context twice. Once for `prfInsert`, and again in the function to calculate the new context. For each assumption we make in the type about the context we must ensure it holds for subsequent steps. Unfortunately, this can result in verbose type signatures.

Devices are linked together using `Link`. For devices to be connected we must assert, using `CanConnect` (Figure 29a), that they have free connections left to make. Like `Insert` we must also update the context for each assertion we make about each devices state. With this definition of `Link` we make no restrictions on linking devices to themselves.

4.2.3 Handler for the Bigraph EDSL

Given the algebraic notation for bigraphs their representation as an algebraic data type naturally follows. Figure 30 presents our bigraph implementation. Entities are a simple data structure capturing the arity of the entity and a unique identifier. `Entity` is parameterised by the sort type as a value. Although, we can use the arity of an entity to inform the length of a `Vect` instance to capture the link graph we must remember that bigraph’s are constructed by interpretation of an instance of `Wireless`. Interpretation must ensure that the construction of the place graph correctly matches the description from the specification, and that the entities used in the place graph are embedded correctly within the final version of the link graph. The final state of the link and place graphs will not be known until we end the specification. Therefore we must delay construction of the bigraph model until then. Thus, interpretation of `Wireless` specifications will return an intermediate bigraph representation used for constructing the algebraic bigraph representation. This is the representation presented in Figure 30.

The type, `Bigraph`, is indexed by the concrete type describing the bigraph’s “types” and specification of the place graph follows the algebraic bigraph definition. Entity arity nor the link graph are described within the `Node` constructor. Borrowing from existing algebraic graph definitions [41] links and external names are represented using `Connect` and `Outside`. `Overlay` describes the union of two bigraph descriptions into a single bigraph. Construction of a more compact algebraic bigraph model from `Bigraph` is not described here.

```

data Entity : (type : Type) -> (value : type) -> Type where
  MkEntity : (n : Nat) -> (arity : Nat) -> Entity type value

data BiGraph a = Identity | Node (Entity a value)
  | Nest (BiGraph a) (BiGraph a) | Merge (BiGraph a) (BiGraph a)
  | Par (BiGraph a) (BiGraph a)
  | Connect (BiGraph a) (BiGraph a) | Outside String
  | Overlay (BiGraph a) (BiGraph a)

```

■ **Figure 30** Naïve Algebraic Representation of a Bigraph.

Figure 31 presents the complete instance of `Handler` for `Wireless`. We use the accumulator of `Handler` to capture the bigraph instance being constructed, and a counter to generate fresh identifiers. An instance of `RealVar` translates variables into `Entity` instances, the type `Ty` has been reused for entity types. Variables are turned into entities and extend the environment for each new variable definition. Insertion of entities creates a `Nest` instruction and each variable definition is inserted into a `Node` constructor. Although the definition of the handler for `Insert` looks repetitive, dependent pattern matching on the side-condition (`prfValid`) is required to ensure that the correct proofs are considered at the type level [38]. Each lookup and proof used for each case have different types. Further, for each operation that updates the context we must also update the environment accordingly. With the updates to the environment mirroring the updates made at the type level. The `Overlay` instruction combines the existing bigraph (`acc`) with a new nesting of parent and child. Interpretation of `Link` follows that for `Insert` by updating the environment for each change in the context, and appends to the accumulated bigraph using `Overlay`, a new edge in the link graph using `Connect`.

4.2.4 Example Bigraph Instances

Figure 32 illustrates how several example bigraphs can be specified using `Wireless`. The type-synonym `WirelessDesc` sets the expected initial and end states – cf. `Files` in Section 4.1. Figure 32a replicates the example from Figure 24. Figure 32b presents a failing example that will not type-check as the domain model specifies that models only support two connections. The final `link` expression will fail to type-check as the abstract state for `mobileA` will have decremented the number of free connections to zero.

4.3 Exemplar 3: Global Session Descriptions

Multi-Party Session Types (MPST) are a typing discipline that allows formal protocol narrations to dictate the type checking process such that implementations of the protocol are *known* to adhere to a given formal narration [30]. Global session types present an overview of the interactions made between entities, and entities have local types that describes their known interactions. Existing work has seen to extend MPST implementation and theory to support reasoning on message values [30, 61, 6]. When looking to realise global session types in a dependently typed language care must be taken that values introduced in the description are used by roles that know about the value.

`SESSIONS` is an EDSL for describing global session descriptions [19]. Figure 33 illustrates how `SESSIONS` can be written using `RESOURCES` as `Sessions`, and Figure 34 details accompanying data types and functions. For brevity, we have not included the creation of value dependent messages. We have, however, extended the EDSL with expressions to reason explicitly about channels, that borrows from existing work [31]. The type of `Sessions` is

```

RealVar Ty where
  CalcRealType ROOM      = Entity Ty ROOM
  CalcRealType BLDG      = Entity Ty BLDG
  CalcRealType (DEVICE x) = Entity Ty (DEVICE x)
  CalcRealType MSG       = Entity Ty MSG

Handler Ty CalcStateType Wireless (Nat, BiGraph Ty) (Basics.id) where
  handle env NewBuilding (ctr,g) cont =
    cont MkVar (MkTag (MkEntity ctr Z)::env) (S ctr,g)
  handle env NewRoom (ctr,g) cont = cont MkVar (MkTag (MkEntity ctr Z)::env) (S ctr,g)
  handle env (NewDevice ty) (ctr,g) cont =
    cont MkVar (MkTag (MkEntity ctr (maxConn ty))::env) (S ctr,g)
  handle env NewMessage (ctr,g) cont = cont MkVar (MkTag (MkEntity ctr Z)::env) (S ctr,g)

  handle env (Insert varX varY prfValid prfFree prfInsert) (ctr,g) cont with (prfValid)
    handle env (Insert varX varY prfValid prfFree prfInsert) (ctr,g) cont | ValidBR = do
      let MkTag rm = lookup env prfFree
          let env' = (update env prfFree Use)
          let MkTag bld = lookup env' prfInsert
          let env'' = update env' prfInsert (Assign varX ValidBR)
          cont () env'' (ctr,Overlay (Nest (Node bld) (Node rm)) g)
    handle env (Insert varX varY prfValid prfFree prfInsert) (ctr,g) cont | ValidRD = do
      let MkTag dev = lookup env prfFree
          let env' = (update env prfFree Use)
          let MkTag rm = lookup env' prfInsert
          let env'' = update env' prfInsert (Assign varX ValidRD)
          cont () env'' (ctr,Overlay (Nest (Node rm) (Node dev)) g)
    handle env (Insert varX varY prfValid prfFree prfInsert) (ctr,g) cont | ValidDM = do
      let MkTag msg = lookup env prfFree
          let env' = (update env prfFree Use)
          let MkTag dev = lookup env' prfInsert
          let env'' = update env' prfInsert (Assign varX ValidDM)
          cont () env'' (ctr,Overlay (Nest (Node dev) (Node msg)) g)

  handle env (Link varX varY prfSpaceX prfSpaceY) (ctr,g) cont = do
    let MkTag x = lookup env prfSpaceX
        let env' = update env prfSpaceX Connect
        let MkTag y = lookup env' prfSpaceY
        let env'' = update env' prfSpaceY Connect
        cont () env'' (ctr,Overlay (Connect (Node x) (Node y)) g)

  handle env End (ctr,g) cont = cont () Nil (ctr,g)

```

■ **Figure 31** Handler instance for `Wireless`.

further parameterised by a list of participants in the protocol, allowing the EDSL to utilise this information for each expression. `Sessions` expressions include message creation, channel construction and destruction, sending of messages, allowing access to message values, and termination of session descriptions.

Central to the operation of `Sessions` is reasoning about the abstract state associated with messages and communication channels. Messages have metatype `DATA` capturing the type of the message, and an associated state listing the actors aware of the message. Channels have metatype `CHAN` capturing the involved actors, and an associated state denoting the connection state: `Bound` or `Free`. The function `CalcStateType` maps meta types to concrete state types.

The construction of `Sessions` follows that of `SESSIONS` but in a more general framework. Message creation extends the list of state with a new abstract state asserting that the creator `a` knows of the message, and the expression's use is restricted to actors listed in the descriptions type. Similarly channel creation extends the list of states with a new abstract state asserting the channel is `Bound`, and restricts channel creation to actors listed in the description's type.


```

example : WirelessDesc m
example = do
  buildingA <- newBuilding
  buildingB <- newBuilding

  roomA <- newRoom
  roomB <- newRoom
  roomC <- newRoom

  laptopA <- newDevice LAPTOP
  laptopB <- newDevice LAPTOP
  laptopC <- newDevice LAPTOP
  mobile <- newDevice MOBILE

  msg <- newMessage

  insert roomA buildingA
  insert laptopA roomA
  insert mobile roomA
  insert msg mobile
  insert roomB buildingA
  insert laptopB roomB
  insert roomC buildingB
  insert laptopC roomC

  link laptopA laptopC
  link mobile laptopB

end

```

```

example : WirelessDesc m
example = do
  buildingA <- newBuilding
  roomA <- newRoom

  insert roomA buildingA

  mobileA <- newDevice MOBILE
  mobileB <- newDevice MOBILE
  mobileC <- newDevice MOBILE
  mobileD <- newDevice MOBILE

  insert mobileA roomA
  insert mobileB roomA
  insert mobileC roomA
  insert mobileD roomA

  link mobileA mobileB
  link mobileA mobileC
  link mobileA mobileD
end

```

(a) Example Bigraph from Figure 24.

(b) Example Failing Bigraph.

■ **Figure 32** Example Specifications using `Wireless`.

This specification implies that we are free to make connections between any two actors in `ps`. We could add a predicate to `Sessions` that restricts channel creation to specific pairings of actors. Closing a channel changes the channel’s abstract state to `Free`. Sending messages along a channel requires an active channel guaranteed by the predicate `ChannelHasState`, and proof (using `KnowsData`) that the sender (`s`) knows about the message. Once a message has been sent the abstract state of the message is updated to reflect that the receiver is now aware of the message.

Figure 35 presents the definition for these and other predicates used in `Sessions`. The expression `ReadMsg` facilitates reasoning using message values that are known to all participants. Session descriptions conclude if all abstract states are in a valid end state. For messages, this is immaterial and for connections they must have been closed.

It is reasonable to assume that we can define a projection function as an instance of `Handler`. The type for `handle`, however, requires that we build a continuation that can be applied to a value associated with the expression. When projecting global types in a multi-party session some expressions are irrelevant if the role being projected for is not involved [14]. The type for `handle` is too constrained for `Sessions` implementation. Future work will be to investigate how a projection function for `Sessions` can be constructed.

Figure 36 present several example session descriptions. The function `Session` is a type-synonym to restrict the starting and ending type-level context to `Nil`. Figure 36a models the salient aspects of the TCP handshake [51]. Here Alice and Bob establish a channel, and Alice sends to Bob a sequence number (`x`) that Bob must return incremented by one. Similarly, Bob sends Alice a sequence (`y`) that Alice must return incremented by one. In our description we use dependent pairs to reason about the message contents.

20:22 A Framework for Resource Dependent EDSLs in a Dependently Typed Language

```

data Sessions : (participants : List Actor) -> Lang Ty CalcStateType where

NewData : (a : Actor) -> (type : Type) -> (prf : Elem a ps)
  -> Sessions ps (Var Ty (DATA type)) old
  (\lbl => MkStateItem (DATA type) lbl (MkDataState [a] type) :: old)

NewConnection : (a,b : Actor) -> (prfS : Elem a ps) -> (prfR : Elem b ps)
  -> Sessions ps (Var Ty (CHAN (a,b))) old
  (\lbl => MkStateItem (CHAN (a,b)) lbl (MkChanState Bound) :: old)

EndConnection : (chan : Var Ty (CHAN (a,b)))
  -> (prf : InContext (CHAN (a,b)) (ChannelHasState Bound chan) old)
  -> Sessions ps () old (const $ update old prf (SetChannelState Free))

SendLeft : (chan : Var Ty (CHAN (s,r)))
  -> (msg : Var Ty (DATA type))
  -> (prfActive : InContext (CHAN (s,r)) (ChannelHasState Bound chan) old)
  -> (prfKnows : InContext (DATA type) (KnowsData s msg) old)
  -> Sessions ps () old (const $ update old prfKnows (ExpandWhoKnows r))

SendRight : (chan : Var Ty (CHAN (s,r)))
  -> (msg : Var Ty (DATA type))
  -> (prfActive : InContext (CHAN (s,r)) (ChannelHasState Bound chan) old)
  -> (prfKnows : InContext (DATA type) (KnowsData r msg) old)
  -> Sessions ps () old (const $ update old prfKnows (ExpandWhoKnows s))

ReadMsg : (msg : Var Ty (DATA type))
  -> (prf : InContext (DATA type) (AllKnow ps msg) old)
  -> Sessions ps type old (const old)

StopSession : AllContext EndState old -> Sessions ps () old (const Nil)

```

■ **Figure 33** An EDSL for describing Global Multi-Party Session Types.

<pre> data Actor = MkActor String data Usage = Free Bound </pre> <p>(a) Actors, Usage, and Metatypes.</p>	<pre> data ChanState = MkChanState Usage data DataState = MkDataState (List Actor) Type </pre> <p>(b) Abstract States.</p>
<pre> data Ty = CHAN (Actor, Actor) DATA Type </pre> <p>(c) Function to compute state types.</p>	<pre> CalcStateType : Ty -> Type CalcStateType (CHAN _) = ChanState CalcStateType (DATA _) = DataState </pre>

■ **Figure 34** Core accompanying data types and functions.

Figures 36b and 36c present two examples that fail to type-check. The first Figure 36b demonstrates how sending on the wrong channel will result in a type error. Here Alice is not involved in the communication between Bob and Charlie. The second example Figure 36c shows an example that will fail as the message (`m`) is not yet known by all participants.

5 Related Work

The implementation of RESOURCES builds upon existing techniques developed for Effects [11] that realise well studied theoretical models [1, 43, 50]. These models were realised in a dependently typed language using straightforward idiomatic constructs: Hoare monads as a parameterised data type; and algebraic effect handlers using interfaces.

```

data ChannelHasState : (assumedState : Usage)
  -> (chan : Var Ty (CHAN (s,r)))
  -> (actual : StateItem Ty CalcStateType (CHAN (s,r)))
  -> Type where
  ChanHasState : ChannelHasState st ch (MkStateItem (CHAN (s,r)) ch (MkChanState st))

```

(a) Asserting Channel State.

```

data AllKnow : (as : List Actor)
  -> (var : Var Ty (DATA type))
  -> (item : StateItem Ty CalcStateType (DATA ty))
  -> Type where
  NilKnows : (prf : Elem x as)
  -> AllKnow [x] msg (MkStateItem (DATA ty) msg (MkDataState as ty))
  ConsKnows : (prf : Elem x as)
  -> (later : AllKnow xs msg (MkStateItem (DATA type) msg (MkDataState as ty)))
  -> AllKnow (x::xs) msg (MkStateItem (DATA type) msg (MkDataState as ty))

```

(b) Asserting that all participants know a value.

```

data KnowsData : (actor : Actor)
  -> (var : Var Ty (DATA type))
  -> (item : StateItem Ty CalcStateType (DATA type))
  -> Type where
  DoesKnow : (prf : x = y) -> (prfE : Elem x actors)
  -> KnowsData y var (MkStateItem (DATA type) var (MkDataState actors type))

```

(c) Asserting that a participant know a value.

```

data EndState : (ty : Ty) -> StateItem Ty CalcStateType ty -> Type where
  EndData : EndState (DATA type) state
  EndConn : EndState (CHAN (s,r)) (MkStateItem (CHAN (s,r)) lbl (MkChanState Free))

```

(d) Asserting final end states.

■ **Figure 35** Predicates used in `Sessions`.

5.1 Theoretical-Oriented Approaches

First we examine other theoretical approaches to realising substructural type-systems for EDSLs that use expressive logics as a base formalism.

Hoare Type Theory. *Hoare Type Theory* [43] has been used to describe programs with substructural type systems [7]. Here types are associated with Hoare triples that are translated to refinement types [23, 27] to ensure triple satisfaction. Ynot is an extension of the Coq proof assistant to provide reasoning about programs using Hoare Type Theory [42]. Similar work has presented a variant of the State Monad that provides Hoare style reasoning on the captured state [60]. Our approach also utilises Hoare triples but not to reason about individual types *per se*, but rather about the entire type-level state of our program. This is much similar to existing work [8, 7] in which the authors were restricted to reasoning about the program’s state, described as a state monad, in its entirety. Use of quantifiers over our abstract state allows us to reason about specific aspects of a program’s state.

Typestates. *Typestates* [56, 22, 5] have been shown to provide a formal basis for building substructural type-systems [39]. Using their approach the authors also show how to incorporate behavioural typing [16] as well. Here each type in their formalism is associated with a type-level state and value level operations apply, at the type level, state transitions to the modelled state. This is a more formal treatment compared to our approach, however, we acknowledge the similarity in associating types with a type-level state. Rather than use typestates as a base formalism we utilise parameterised monads.

```

TCPHandshake : Session m [Alice, Bob]
TCPHandshake = do
  chan <- setup Alice Bob

  m1 <- msg Alice (Packet, Nat)
  sendLeft chan m1

  (p,x) <- read m1
  m2 <- msg Bob
    (Packet, (x' ** x' = S x),
             Nat)

  sendRight chan m2

  (p,xplus,y) <- read m2

  m3 <- msg Alice
    (Packet, (x' ** x' = S x),
             (y' ** y' = S y))

  sendLeft chan m3
  destroy chan
end
(a) TCP Handshake.

WrongChan : Session m [Alice,Bob,Charlie]
WrongChan = do
  chan <- setup Alice Bob
  net <- setup Bob Charlie

  m <- msg Alice String
  sendRight net m
  ?end
(b) Sending on a wrong channel.

UnableToRead : Session m
               [Alice,Bob,Charlie]
UnableToRead = do
  chan <- setup Alice Bob

  m <- msg Alice String
  sendLeft chan m

  val <- read m
  ?end
(c) Invalid Read Access.

```

■ **Figure 36** Example Global Session Descriptions.

Separation Logics. *Separation logic* [45] has been used to provide another formal treatment towards customising standard substructural type-systems with custom resources [34]. This work supports customisation of resources, and controls on said resource to be specified on a *per-module* or *per-library* basis. Similarly, the authors use state-transition systems by way of commutative monoids, to reason about substructural properties.

Very recent work has investigated the use of separation logics to build intrinsically-typed definitional interpreters for linear/session-typed languages [52]. Developed in Agda [44] the authors present a collection of reusable and composable abstractions to support interpreter construction. Our approach differs in that we ground our work in hoare logic and provide a singular unified framework to capture common language expressions common to all EDSLs, and integrated support for reasoning on abstract program state. We will carefully study the use of separation logics and see how RESOURCES can be bettered. Of interest will be the ability of the author’s approach to realise global MPST.

Quantitative Type-Systems. Substructural Type-Systems [65] support various different styles of reasoning about variable usage at the type level. Linear typing providing *exactly once* semantics [64, 63], and Affine systems *at most once* [62, 15]. Generally speaking, *Quantitative Type-Theory* (QTT) [2] provides a more general framework to reason about resource usage. However it is not clear how state-based substructural properties (cf. nesting for Bigraphs – Section 4.2.1) can be modelled within QTT. Their substructural properties are not all about quantitative usage. Regardless, QTT is a promising direction for reasoning about quantitative resource usage.

5.2 Practical-Oriented Approaches

RESOURCES is highly dependent on Idris specific features such as interfaces and proof search, as well as dependent types. Realising RESOURCES in dissimilar languages would require more complicated work arounds to realise a similar framework, or require direct modification to the compiler. Like Effects, RESOURCES is a *plain-old-library* that does not require any

language extensions, and leverages a language in which dependent types were not retrofitted. This gives us static compile time guarantees that our framework, and thus our EDSLs, are well-typed. We now examine other practical approaches that involve expressive host languages which support construction of bespoke substructural type-systems within the host language itself.

Substructural Type-Systems. There exist several general purpose languages that provide full or experimental support for substructural type-systems. Linear typing has been realised for ATS [55]. Clean has implemented uniqueness typing [21] that influenced Rust’s ownership types [35, 36]. An extension is being developed for Haskell that leverages linearity for correct variable usage [4]. Interestingly, this extension also uses a parameterised type to replicate typestates when reasoning about a socket example. Idris itself has experimental support for linear typing [37], and Clean’s Uniqueness types [14]. Future iterations of Idris³, however, will support Quantitative Type-Theory [2]. As we described in Section 5.1, it is not clear how we can use these quantitative systems to describe state-based substructural properties.

Expressive Type-Systems. Construction of domain specific substructural type systems for EDSLs can be achieved using other as expressive non-dependently typed host languages. Racket is a general purpose language that supports EDSL creation through fine-grained control over the language’s type-system [25]. The original version of F^* was a general purpose language with value-dependent types [57, 58]. Whereas Idris provides full-spectrum dependent types, F^* provides value-dependencies using refinement types. F^* was extended to provide better support for dependently typed and effectful programming [59]. Such languages provide novel, alternate, environment in which to construct “value-dependently-typed” programs. How the approach behind RESOURCES is transferable to these languages is worth investigating.

Dependent-Type Systems. Although, the framework has been realised using Idris the techniques presented are agnostic to dependently typed languages. Any other dependently typed language that supports full-spectrum dependent types, such as Agda [44], will be suitable for implementing the ideas. It has been shown how Dependent Haskell [66] can realise Idris’ Effects library [24, § 3.2.3]. Existing work has investigated embedding linear type-systems for EDSLs into Haskell [49]. In their implementation the author’s make extensive use of Haskell’s typeclass mechanism, a *Higher Order Abstract Syntax* embedding, and Dependent Haskell [48].

ST is an improvement upon Effects by not only associating resources with variables, but facilitating vertical and horizontal effect composition [12]. ST is a resource dependent EDSL, and makes extensive use of Idris’ Interface mechanism for effect definition. RESOURCES sits in between these two implementations, borrowing the algebraic language definition from Effects and associating abstract state with variables from ST. We position RESOURCES as a framework for defining EDSLs with domain specific substructural type-systems. ST and Effects are general purpose.

³ <https://github.com/edwinb/idris2>

6 Future Work

RESOURCES is a promising framework for constructing EDSLs with interesting type-systems. However, there are a few limitations to our current approach.

Each EDSL requires that the complete set of resource types to be used be known at EDSL design time. Our EDSL are closed worlds. This impacts upon the composability of resources between EDSL instances. For example, state resources have to be created per EDSL. This limitations originate from the framework's design. It was not designed for resource reuse in mind. A promising direction will be to look at how Idris' interfaces can look to better the specification and use of resources in EDSLs.

Similarly, our type-level state holds too much information. If we were to call out to sub-programs we must be careful about the effect that the resulting state (of the callee program) has on the caller program. We see this in the design of `Files` (Section 4.1) when passing around file handles we need to ensure that closed files remain closed. While one can state that sub-programs are closed it will be interesting to investigate how to deal with interactions of states between programs. When looking at program composition, and resource reuse between EDSLs, how we interact with type-level state is important. Our use of Hoare logics prohibits the inspection, individually, of resources in isolation. By basing the framework on separation logics rather than Hoare triples, we can look to address these limitations.

Updating the type-level context multiple times in a type signature, can lead to a more verbose style of type-level programming. For example, consider the type-signatures for `Link` and `Insert` for `Wireless` in Section 4.2.2. The limitation here is Idris' own syntax: type signatures are not equivalent to a function body. Future work will be to see how we can reason better about the transformations made to the abstract state within a type signature.

7 Conclusions

RESOURCES has been developed to explore construction of EDSLs with substructural type-systems supporting autonomic management of domain specific abstract resources and type-level reasoning on such resources. Idris' support for *auto-implicit* arguments allows languages to be presented cleanly, where proofs that properties hold are hidden but present during type-checking. Resources and their state need not be listed explicitly at the type-level.

We have demonstrated the use of RESOURCES through construction of several exemplar EDSLs. Type-level predicates provided compile time guarantees over various substructural properties. Providing static compile time checks that correct EDSL instances are constructed. We have demonstrated how `Handler` instances can: run interactive programs – Section 4.1.2; and construct data types – Section 4.2.3. When we construct data structures, however, the correctness-by-construction guarantees are not necessarily carried over. It will be interesting to see how we can use RESOURCES to do so. This would be useful for our Bigraph example. This is future work. Further, we have seen when the `Handler` interface was not enough for our needs (Section 4.3), and noted limitations on program and resource composition – Section 6.

We are using RESOURCES to develop EDSLs for reasoning about the structural and behavioural aspects of System-on-a-Chip Designs. Within these languages a substructural type-system allows one to constrain expressions using type-level resources derived from finite sources and behavioural specifications. Ensuring, for example, that ports can be connected to only once, and that interfaces and connections are well-formed respective to a given specification. RESOURCES helps by providing a common framework to explore different model designs without specifying the same boilerplate again and again.

References

- 1 Robert Atkey. Parameterised notions of computation. *J. Funct. Program.*, 19(3-4):335–376, 2009. doi:10.1017/S095679680900728X.
- 2 Robert Atkey. Syntax and semantics of quantitative type theory. In Anuj Dawar and Erich Grädel, editors, *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2018, Oxford, UK, July 09-12, 2018*, pages 56–65. ACM, 2018. doi:10.1145/3209108.3209189.
- 3 Lennart Augustsson and Magnus Carlsson. An Exercise in Dependent Types: A Well-Typed Interpreter. In *In Workshop on Dependent Types in Programming, Gothenburg*, 1999.
- 4 Jean-Philippe Bernardy, Mathieu Boespflug, Ryan R. Newton, Simon Peyton Jones, and Arnaud Spiwack. Linear haskell: practical linearity in a higher-order polymorphic language. *PACMPL*, 2(POPL):5:1–5:29, 2018. doi:10.1145/3158093.
- 5 Kevin Bierhoff and Jonathan Aldrich. Modular typestate checking of aliased objects. In Richard P. Gabriel, David F. Bacon, Cristina Videira Lopes, and Guy L. Steele Jr., editors, *Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2007, October 21-25, 2007, Montreal, Quebec, Canada*, pages 301–320. ACM, 2007. doi:10.1145/1297027.1297050.
- 6 Laura Bocchi, Kohei Honda, Emilio Tuosto, and Nobuko Yoshida. A theory of design-by-contract for distributed multiparty interactions. In *CONCUR 2010 - Concurrency Theory, 21th International Conference, CONCUR 2010, Paris, France, August 31-September 3, 2010. Proceedings*, pages 162–176, 2010. doi:10.1007/978-3-642-15375-4_12.
- 7 Johannes Borgström, Juan Chen, and Nikhil Swamy. Verifying stateful programs with substructural state and hoare types. In Ranjit Jhala and Wouter Swierstra, editors, *Proceedings of the 5th ACM Workshop Programming Languages meets Program Verification, PLPV 2011, Austin, TX, USA, January 29, 2011*, pages 15–26. ACM, 2011. doi:10.1145/1929529.1929532.
- 8 Johannes Borgström, Andrew D. Gordon, and Riccardo Pucella. Roles, stacks, histories: A triple for hoare. *J. Funct. Program.*, 21(2):159–207, 2011. doi:10.1017/S0956796810000134.
- 9 Edwin Brady. Idris, a general-purpose dependently typed programming language: Design and implementation. *J. Funct. Program.*, 23(5):552–593, 2013. doi:10.1017/S095679681300018X.
- 10 Edwin Brady. Programming and reasoning with algebraic effects and dependent types. In Greg Morrisett and Tarmo Uustalu, editors, *ACM SIGPLAN International Conference on Functional Programming, ICFP’13, Boston, MA, USA - September 25 - 27, 2013*, pages 133–144. ACM, 2013. doi:10.1145/2500365.2500581.
- 11 Edwin Brady. Resource-dependent algebraic effects. In Jurriaan Hage and Jay McCarthy, editors, *Trends in Functional Programming - 15th International Symposium, TFP 2014, Soesterberg, The Netherlands, May 26-28, 2014. Revised Selected Papers*, volume 8843 of *Lecture Notes in Computer Science*, pages 18–33. Springer, 2014. doi:10.1007/978-3-319-14675-1_2.
- 12 Edwin Brady. State Machines All The Way Down: An Architecture for Dependently Typed Applications. Unpublished Draft., 2016.
- 13 Edwin Brady. *Type-Driven Development with Idris*. Manning, 1st edition, 2016.
- 14 Edwin Brady. Type-driven development of concurrent communicating systems. *Computer Science (AGH)*, 18(3), 2017. doi:10.7494/csci.2017.18.3.1413.
- 15 Michele Bugliesi, Stefano Calzavara, Fabienne Eigner, and Matteo Maffei. Affine refinement types for secure distributed programming. *ACM Trans. Program. Lang. Syst.*, 37(4):11:1–11:66, 2015. doi:10.1145/2743018.
- 16 Luís Caires and João Costa Seco. The type discipline of behavioral separation. In Roberto Giacobazzi and Radhia Cousot, editors, *The 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL ’13, Rome, Italy - January 23 - 25, 2013*, pages 275–286. ACM, 2013. doi:10.1145/2429069.2429103.
- 17 Elias Castegren and Tobias Wrigstad. Reference capabilities for concurrency control. In Shriram Krishnamurthi and Benjamin S. Lerner, editors, *30th European Conference on Object-Oriented Programming, ECOOP 2016, July 18-22, 2016, Rome, Italy*, volume 56 of *LIPICs*, pages 5:1–5:26. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2016. doi:10.4230/LIPICs.ECOOP.2016.5.

- 18 N.G de Bruijn. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the church-rosser theorem. *Indagationes Mathematicae (Proceedings)*, 75(5):381–392, 1972.
- 19 Jan de Muijnck-Hughes, Edwin Brady, and Wim Vanderbauwhede. Value-dependent session design in a dependently typed language. In Francisco Martins and Dominic Orchard, editors, *Proceedings Programming Language Approaches to Concurrency- and Communication-cEntric Software, PLACES@ETAPS 2019, Prague, Czech Republic, 7th April 2019*, volume 291 of *EPTCS*, pages 47–59, 2019. doi:10.4204/EPTCS.291.5.
- 20 Jan de Muijnck-Hughes and Wim Vanderbauwhede. A typing discipline for hardware interfaces. In Alastair F. Donaldson, editor, *33rd European Conference on Object-Oriented Programming, ECOOP 2019, July 15-19, 2019, London, United Kingdom*, volume 134 of *LIPICs*, pages 6:1–6:27. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019. doi:10.4230/LIPICs.ECOOP.2019.6.
- 21 Edsko de Vries, Rinus Plasmeijer, and David M. Abrahamson. Uniqueness typing simplified. In Olaf Chitil, Zoltán Horváth, and Viktória Zsók, editors, *Implementation and Application of Functional Languages, 19th International Workshop, IFL 2007, Freiburg, Germany, September 27-29, 2007. Revised Selected Papers*, volume 5083 of *Lecture Notes in Computer Science*, pages 201–218. Springer, 2007. doi:10.1007/978-3-540-85373-2_12.
- 22 Robert DeLine and Manuel Fähndrich. Typestates for objects. In Martin Odersky, editor, *ECOOP 2004 - Object-Oriented Programming, 18th European Conference, Oslo, Norway, June 14-18, 2004, Proceedings*, volume 3086 of *Lecture Notes in Computer Science*, pages 465–490. Springer, 2004. doi:10.1007/978-3-540-24851-4_21.
- 23 Ewen Denney. Refinement types for specification. In David Gries and Willem P. de Roever, editors, *Programming Concepts and Methods, IFIP TC2/WG2.2,2.3 International Conference on Programming Concepts and Methods (PROCOMET '98) 8-12 June 1998, Shelter Island, New York, USA*, volume 125 of *IFIP Conference Proceedings*, pages 148–166. Chapman & Hall, 1998.
- 24 Richard A. Eisenberg. Dependent types in haskell: Theory and practice. *CoRR*, abs/1610.07978, 2016. arXiv:1610.07978.
- 25 Matthias Felleisen, Robert Bruce Findler, Matthew Flatt, Shriram Krishnamurthi, Eli Barzilay, Jay A. McCarthy, and Sam Tobin-Hochstadt. A programmable programming language. *Commun. ACM*, 61(3):62–71, 2018. doi:10.1145/3127323.
- 26 Martin Fowler. *Domain-Specific Languages*. Addison-Wesley Signature Series. Addison-Wesley Professional, 1 edition, October 2010.
- 27 Timothy S. Freeman and Frank Pfenning. Refinement types for ML. In David S. Wise, editor, *Proceedings of the ACM SIGPLAN'91 Conference on Programming Language Design and Implementation (PLDI), Toronto, Ontario, Canada, June 26-28, 1991*, pages 268–277. ACM, 1991. doi:10.1145/113445.113468.
- 28 Colin S. Gordon, Matthew J. Parkinson, Jared Parsons, Aleks Bromfield, and Joe Duffy. Uniqueness and reference immutability for safe parallelism. In Gary T. Leavens and Matthew B. Dwyer, editors, *Proceedings of the 27th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2012, part of SPLASH 2012, Tucson, AZ, USA, October 21-25, 2012*, pages 21–40. ACM, 2012. doi:10.1145/2384616.2384619.
- 29 C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, 1969. doi:10.1145/363235.363259.
- 30 Kohei Honda, Nobuko Yoshida, and Marco Carbone. Multiparty asynchronous session types. *J. ACM*, 63(1):9:1–9:67, 2016. doi:10.1145/2827695.
- 31 Raymond Hu and Nobuko Yoshida. Explicit connection actions in multiparty session types. In *Fundamental Approaches to Software Engineering - 20th International Conference, FASE 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings*, pages 116–133, 2017. doi:10.1007/978-3-662-54494-5_7.

- 32 Wei Huang, Werner Dietl, Ana Milanova, and Michael D. Ernst. Inference and checking of object ownership. 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 181–206. Springer, 2012. doi:10.1007/978-3-642-31057-7_9.
- 33 Ohad Kammar, Sam Lindley, and Nicolas Oury. Handlers in action. In Greg Morrisett and Tarmo Uustalu, editors, *ACM SIGPLAN International Conference on Functional Programming, ICFP’13, Boston, MA, USA - September 25 - 27, 2013*, pages 145–158. ACM, 2013. doi:10.1145/2500365.2500590.
- 34 Neelakantan R. Krishnaswami, Aaron Turon, Derek Dreyer, and Deepak Garg. Superficially substructural types. In Peter Thiemann and Robby Bruce Findler, editors, *ACM SIGPLAN International Conference on Functional Programming, ICFP’12, Copenhagen, Denmark, September 9-15, 2012*, pages 41–54. ACM, 2012. doi:10.1145/2364527.2364536.
- 35 Amit A. Levy, Michael P. Andersen, Bradford Campbell, David E. Culler, Prabal Dutta, Branden Ghena, Philip Levis, and Pat Pannuto. Ownership is theft: experiences building an embedded OS in rust. In Shan Lu, editor, *Proceedings of the 8th Workshop on Programming Languages and Operating Systems, PLOS 2015, Monterey, California, USA, October 4, 2015*, pages 21–26. ACM, 2015. doi:10.1145/2818302.2818306.
- 36 Nicholas D. Matsakis and Felix S. Klock II. The rust language. In Michael Feldman and S. Tucker Taft, editors, *Proceedings of the 2014 ACM SIGAda annual conference on High integrity language technology, HILT 2014, Portland, Oregon, USA, October 18-21, 2014*, pages 103–104. ACM, 2014. doi:10.1145/2663171.2663188.
- 37 Conor McBride. I got plenty o’ nuttin’. In Sam Lindley, Conor McBride, Philip W. Trinder, and Donald Sannella, editors, *A List of Successes That Can Change the World - Essays Dedicated to Philip Wadler on the Occasion of His 60th Birthday*, volume 9600 of *Lecture Notes in Computer Science*, pages 207–233. Springer, 2016. doi:10.1007/978-3-319-30936-1_12.
- 38 Conor McBride and James McKinna. The view from the left. *J. Funct. Program.*, 14(1):69–111, 2004. doi:10.1017/S0956796803004829.
- 39 Filipe Militão, Jonathan Aldrich, and Luís Caires. Substructural typestates. In Nils Anders Danielsson and Bart Jacobs, editors, *Proceedings of the 2014 ACM SIGPLAN Workshop on Programming Languages meets Program Verification, PLPV 2014, January 21, 2014, San Diego, California, USA, Co-located with POPL ’14*, pages 15–26. ACM, 2014. doi:10.1145/2541568.2541574.
- 40 Robin Milner. *The Space and Motion of Communicating Agents*. Cambridge University Press, 2009.
- 41 Andrey Mokhov. Algebraic graphs with class (functional pearl). In *Proceedings of the 10th ACM SIGPLAN International Symposium on Haskell, Oxford, United Kingdom, September 7-8, 2017*, pages 2–13, 2017. doi:10.1145/3122955.3122956.
- 42 Aleksandar Nanevski, Greg Morrisett, Avraham Shinnar, Paul Govereau, and Lars Birkedal. Ynot: dependent types for imperative programs. In *Proceeding of the 13th ACM SIGPLAN international conference on Functional programming, ICFP 2008, Victoria, BC, Canada, September 20-28, 2008*, pages 229–240, 2008. doi:10.1145/1411204.1411237.
- 43 Aleksandar Nanevski, J. Gregory Morrisett, and Lars Birkedal. Hoare type theory, polymorphism and separation. *J. Funct. Program.*, 18(5-6):865–911, 2008. doi:10.1017/S0956796808006953.
- 44 Ulf Norell. Dependently typed programming in agda. In Andrew Kennedy and Amal Ahmed, editors, *Proceedings of TLDI’09: 2009 ACM SIGPLAN International Workshop on Types in Languages Design and Implementation, Savannah, GA, USA, January 24, 2009*, pages 1–2. ACM, 2009. doi:10.1145/1481861.1481862.
- 45 Peter W. O’Hearn. Separation logic. *Commun. ACM*, 62(2):86–95, 2019. doi:10.1145/3211968.

- 46 Dominic A. Orchard and Tomas Petricek. Embedding effect systems in haskell. In Wouter Swierstra, editor, *Proceedings of the 2014 ACM SIGPLAN symposium on Haskell, Gothenburg, Sweden, September 4-5, 2014*, pages 13–24. ACM, 2014. doi:10.1145/2633357.2633368.
- 47 Johan Östlund and Tobias Wrigstad. Multiple aggregate entry points for ownership types. 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 156–180. Springer, 2012. doi:10.1007/978-3-642-31057-7_8.
- 48 Jennifer Paykin. *Linear/non-Linear Types for Embedded Domain-Specific Languages*. PhD thesis, University of Pennsylvania, 2018. Publicly Accessible Penn Dissertations. 2752. URL: <https://repository.upenn.edu/edissertations/2752>.
- 49 Jennifer Paykin and Steve Zdancewic. The linearity monad. In *Proceedings of the 10th ACM SIGPLAN International Symposium on Haskell, Oxford, United Kingdom, September 7-8, 2017*, pages 117–132, 2017. doi:10.1145/3122955.3122965.
- 50 Gordon D. Plotkin and Matija Pretnar. Handlers of algebraic effects. In Giuseppe Castagna, editor, *Programming Languages and Systems, 18th European Symposium on Programming, ESOP 2009, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009, York, UK, March 22-29, 2009. Proceedings*, volume 5502 of *Lecture Notes in Computer Science*, pages 80–94. Springer, 2009. doi:10.1007/978-3-642-00590-9_7.
- 51 Jon Postel. Transmission control protocol. *RFC*, 793:1–91, 1981. doi:10.17487/RFC0793.
- 52 Arjen Rouvoet, Casper Bach Poulsen, Robbert Krebbers, and Eelco Visser. Intrinsically-typed definitional interpreters for linear, session-typed languages. In Jasmin Blanchette and Catalin Hritcu, editors, *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2020, New Orleans, LA, USA, January 20-21, 2020*, pages 284–298. ACM, 2020. doi:10.1145/3372885.3373818.
- 53 Michele Sevegnani and Muffy Calder. Bigrapher: Rewriting and analysis engine for bi-graphs. In *Computer Aided Verification - 28th International Conference, CAV 2016, Toronto, ON, Canada, July 17-23, 2016, Proceedings, Part II*, pages 494–501, 2016. doi:10.1007/978-3-319-41540-6_27.
- 54 Michele Sevegnani, Milan Kabác, Muffy Calder, and Julie A. McCann. Modelling and verification of large-scale sensor network infrastructures. In *23rd International Conference on Engineering of Complex Computer Systems, ICECCS 2018, Melbourne, Australia, December 12-14, 2018*, pages 71–81, 2018. doi:10.1109/ICECCS2018.2018.00016.
- 55 Rui Shi and Hongwei Xi. A linear type system for multicore programming in ATS. *Sci. Comput. Program.*, 78(8):1176–1192, 2013. doi:10.1016/j.scico.2012.09.005.
- 56 Robert E. Strom. Mechanisms for compile-time enforcement of security. In *Conference Record of the Tenth Annual ACM Symposium on Principles of Programming Languages, Austin, Texas, USA, January 1983*, pages 276–284, 1983. doi:10.1145/567067.567093.
- 57 Nikhil Swamy, Juan Chen, and Ravi Chugh. Enforcing stateful authorization and information flow policies in fine. In Andrew D. Gordon, editor, *Programming Languages and Systems, 19th European Symposium on Programming, ESOP 2010, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2010, Paphos, Cyprus, March 20-28, 2010. Proceedings*, volume 6012 of *Lecture Notes in Computer Science*, pages 529–549. Springer, 2010. doi:10.1007/978-3-642-11957-6_28.
- 58 Nikhil Swamy, Juan Chen, Cédric Fournet, Pierre-Yves Strub, Karthikeyan Bhargavan, and Jean Yang. Secure distributed programming with value-dependent types. *J. Funct. Program.*, 23(4):402–451, 2013. doi:10.1017/S0956796813000142.
- 59 Nikhil Swamy, Joel Weinberger, Cole Schlesinger, Juan Chen, and Benjamin Livshits. Verifying higher-order programs with the dijkstra monad. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13, Seattle, WA, USA, June 16-19, 2013*, pages 387–398, 2013. doi:10.1145/2491956.2491978.

- 60 Wouter Swierstra. A hoare logic for the state monad. In *Theorem Proving in Higher Order Logics, 22nd International Conference, TPHOLs 2009, Munich, Germany, August 17-20, 2009. Proceedings*, pages 440–451, 2009. doi:10.1007/978-3-642-03359-9_30.
- 61 Bernardo Toninho and Nobuko Yoshida. Certifying data in multiparty session types. *J. Log. Algebraic Methods Program.*, 90:61–83, 2017. doi:10.1016/j.jlamp.2016.11.005.
- 62 Jesse A. Tov and Riccardo Pucella. Practical affine types. In *Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2011, Austin, TX, USA, January 26-28, 2011*, pages 447–458, 2011. doi:10.1145/1926385.1926436.
- 63 Philip Wadler. Linear types can change the world! In Manfred Broy, editor, *Programming concepts and methods: Proceedings of the IFIP Working Group 2.2, 2.3 Working Conference on Programming Concepts and Methods, Sea of Galilee, Israel, 2-5 April, 1990*, page 561. North-Holland, 1990.
- 64 Philip Wadler. Is there a use for linear logic? In Charles Consel and Olivier Danvy, editors, *Proceedings of the Symposium on Partial Evaluation and Semantics-Based Program Manipulation, PEPM'91, Yale University, New Haven, Connecticut, USA, June 17-19, 1991*, pages 255–273. ACM, 1991. doi:10.1145/115865.115894.
- 65 David Walker. *Advanced Topic in Types and Programming Languages*, chapter Substructural Type Systems, pages 3–43. The MIT Press, 2004.
- 66 Stephanie Weirich, Antoine Voizard, Pedro Henrique Avezedo de Amorim, and Richard A. Eisenberg. A specification for dependent types in haskell. *PACMPL*, 1(ICFP):31:1–31:29, 2017. doi:10.1145/3110275.

Data Consistency in Transactional Storage Systems: A Centralised Semantics

Shale Xiong¹

Department of Computing, Imperial College London, United Kingdom
shale.xiong14@ic.ac.uk

Andrea Cerone²

Department of Computing, Imperial College London, United Kingdom
andrea.cerone@ic.ac.uk

Azalea Raad

MPI-SWS, Kaiserslautern, Germany
azalea@mpi-sws.org

Philippa Gardner

Department of Computing, Imperial College London, United Kingdom
p.gardner@ic.ac.uk

Abstract

We introduce an interleaving operational semantics for describing the client-observable behaviour of atomic transactions on distributed key-value stores. Our semantics builds on abstract states comprising centralised, global key-value stores and partial client views. Using our abstract states, we present operational definitions of well-known consistency models in the literature, and prove them to be equivalent to their existing declarative definitions using abstract executions. We explore two applications of our operational framework:

1. verifying that the COPS replicated database and the Clock-SI partitioned database satisfy their consistency models using trace refinement, and
2. proving invariant properties of client programs.

2012 ACM Subject Classification Theory of computation → Operational semantics

Keywords and phrases Operational Semantics, Consistency Models, Transactions, Distributed Key-value Stores

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2020.21

Funding *Shale Xiong*: The Department of Computing, Imperial College London, and EPSRC Fellowship VeTSpec: Verified Trustworthy Software Specification (EP/R034567/1)

Andrea Cerone: EPSRC Programme Grant REMS: Rigorous Engineering for Mainstream Systems (EP/K008528/1), and EPSRC Fellowship VeTSpec: Verified Trustworthy Software Specification (EP/R034567/1)

Azalea Raad: ERC Horizon 2020 Consolidator Grant “RustBelt” (grant agreement no. 683289)

Philippa Gardner: EPSRC Programme Grant REMS: Rigorous Engineering for Mainstream Systems (EP/K008528/1), and EPSRC Fellowship VeTSpec: Verified Trustworthy Software Specification (EP/R034567/1)

1 Introduction

Transactions are the *de facto* synchronisation mechanism in modern distributed databases. To achieve scalability and performance, distributed databases often use weak transactional consistency guarantees known as *consistency models*. Many consistency models were originally

¹ Shale Xiong has moved to Arm Research, shale.xiong@arm.com.

² Andrea Cerone has moved to Football Radar, andrea.cerone@footballradar.com.



© Shale Xiong, Andrea Cerone, Azalea Raad, and Philippa Gardner; licensed under Creative Commons License CC-BY

34th European Conference on Object-Oriented Programming (ECOOP 2020).

Editors: Robert Hirschfeld and Tobias Pape; Article No. 21; pp. 21:1–21:31

Leibniz International Proceedings in Informatics



LIPICs Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

invented by engineers using (some quite informal) definitions specific to particular real-world reference implementations, e.g. [3, 4, 6, 8, 21, 33, 38, 42]. More recently, general definitions of consistency model have been defined independently of particular implementations, either declaratively using execution graphs [1, 9] or operationally using abstract states or execution graphs [16, 27, 35]. Our challenge is to define a general semantics for weak consistency models with which we can both verify reference implementations *and* analyse the behaviour of client programs with respect to a particular consistency model.

The declarative approach for defining consistency models using execution graphs has been substantially studied [1, 9, 11, 12, 14]. In such graphs, nodes describe the read-write sets of atomic transactions and edges describe the known dependencies between transactions. They capture different consistency models by:

1. constructing *candidate executions* of the whole program comprising transactions in which reads may contain arbitrary values; and
2. applying the consistency-model *axioms* to rule out candidate executions deemed invalid by the axioms.

Such axioms may state, for example, that every read is validated by a write that has written the read value. The most well-known execution graphs are dependency graphs [1] and abstract executions [9, 11]. Dependency graphs tend to be used to analyse client programs, e.g. Fekete et al. [23] derived a static analysis checker for a particular weak consistency model called snapshot isolation; Bernardi and Gotsman [7] developed a static analysis checker for several weak consistency models assuming the so-called snapshot property³; and Beillahi et al. [5] developed a tool based on Lipton's reduction theory [31] for checking robustness⁴ properties against snapshot isolation. Abstract executions, on the other hand, tend to be used to verify implementation protocols, e.g. abstract executions are the standard by which many system engineers demonstrate that their protocols satisfy certain consistency models [3, 33, 42]. Execution graphs provide little information about how the state evolves throughout the execution of a program, and therefore seem unsuitable for invariant-based program analysis of client programs.

The operational approach for defining weak consistency models has been much less studied. Crooks et al. [16] introduced a trace semantics over abstract centralised kv-stores, abstracting the behaviour of the underlying concrete distributed kv-stores, in order to capture the consistency models associated with ANSI/SQL isolation levels. They describe the equivalence of several implementation-specific definitions of consistency model in the literature, but their reliance on the total transaction order suggests that it will be difficult to adapt their work to reason about client programs. Kaki et al. [27] provide an operational semantics over an abstract centralised store, again focusing on ANSI/SQL isolation levels. They develop a program logic and prototype tool for reasoning about client programs, but cannot express fundamental weak consistency models. Nagar and Jagannathan [35] introduce an operational semantics based on abstract-execution graphs, focussing on consistency models for distributed transactions. They provide robustness results for client programs using model checking, but their analysis is indirect in that they move back and forth between abstract executions and dependency graphs. All these approaches have their merits. However, none provide a direct state-based operational semantics for distributed atomic transactions with which to verify distributed implementations and analyse client programs using the usual weak consistency models; see Section 1.1 for further details on this related work.

³ The *snapshot property*, also known as *atomic visibility*, states that transactional reads appear to read from an atomic snapshot of the database and transactional writes appear to commit atomically, i.e. intermediate transactional states are not observable by clients, even if the underlying distributed protocol has a more fine-grained behaviour.

⁴ A particular program (or set of programs) behaves as if the consistency model is serialisability

We introduce an interleaving operational semantics for describing the client-observable behaviour of atomic transactions updating distributed key-value stores (Section 3). Our semantics is based on a notion of abstract states comprising a *centralised key-value store* (kv-store) with multi-versioning and a *client view*. Kv-stores are *global* in that they record all versions of a key; by contrast, client views are *partial* in that a client may see only a subset of the versions. Our client views are partly inspired by the views in the “promising” C11 semantics [28]. An execution step depends simply on the abstract state, the read-write set of the atomic transaction, and an *execution test*, determining if a client with a given view can commit a transaction. Different execution tests give rise to different consistency models, which we show to be equivalent to well-known declarative definitions of consistency models based on abstract executions (reported here and proven in [46]) and thus those based on dependency graphs [14]. Our execution tests are analogous to the commit tests in [16], except that [16] requires analysing the whole trace rather than just the current abstract state.

As in [16, 27, 35], we assume that transactions satisfy the *last-write-wins* resolution policy, a policy widely used in many real-world distributed kv-stores. This means that when a transaction observes several updates to a key, the atomic snapshot contains the value written by the last update. We also assume that our transactions satisfy the *snapshot property*. This is a common assumption in distributed transactional databases, e.g. in online shopping applications, a client only sees one snapshot of the database and only has knowledge that their transaction has successfully committed. The work in [35] also assumes the snapshot property, whereas [16] and [27] do not as their focus is on ANSI/SQL isolation levels [6]. Our execution tests uniformly capture many well-known consistency models (Section 4) including *causal consistency* (CC) [9, 33, 40], *parallel snapshot isolation* (PSI) [3, 42], *snapshot isolation* (SI) [6] and *serialisability* (SER) [37]. The work in [35] is as expressive as our work here; by contrast, [16] is more expressive, capturing e.g. the *read committed* consistency model [6], while [27] is less expressive, capturing SI but not PSI.

Using our operational semantics, we verify that database protocols satisfy their expected consistency models and prove invariant properties of client programs under such consistency models (Section 5). Specifically, we prove the correctness of two database protocols using our general definitions: the COPS protocol for fully replicated kv-stores [33] which satisfies CC (reported in Section 5.1 and proved in [46]), and the Clock-SI protocol for partitioned kv-stores [21] which satisfies SI (given in [46]). These results had been previously shown for specific consistency definitions devised for the specific reference implementations under consideration. We also prove invariant properties of library clients (Section 5.2): the robustness of the single-counter library against PSI, the robustness of the multi-counter library and the banking library [2] against SI, and the mutual exclusion of a lock library against PSI. We believe our robustness results are the first to take into account client sessions: with sessions, we show that multiple counters *are not* robust against PSI. Interestingly, without sessions, Bernardi and Gotsman [7] show that multiple counters *are* robust against PSI using static-analysis techniques which are known not to be applicable to sessions. These results indicate that our operational semantics provides an interesting abstract interface between distributed databases and clients. This was an important goal for us, resonating with recent work that does just this for standard shared-memory concurrency [17, 19, 25, 36].

1.1 Related Work

Operational semantics for defining weak consistency models for distributed atomic transactions have hardly been studied. To our knowledge, the key papers are [16, 35, 27]. We also mention the log-based semantics of Koskinen and Parkinson [29], which only focuses on serialisability but has some resonance with our work.

Crooks et al. [16] proposed a state-based trace semantics for describing weak consistency models that employs concepts similar to our client views and execution tests, called read states and commit tests respectively. In their semantics, a one-step trace reduction is determined by the entire previous history of the trace. By contrast, our reduction step only depends on the current kv-store and client view. They capture more consistency models than us, e.g. *read committed*, because they do not assume the snapshot property due to their focus on ANSI/SQL isolation levels. They use their semantics to demonstrate that several definitions of snapshot isolation given in the literature [6, 18, 22] in fact collapse into one. They do not verify protocol implementations and do not prove invariant properties of client programs. We believe [16] can be used to verify implementations. We believe it might be difficult to use [16] to prove invariant properties of client programs since their commit tests use total traces. In contrast, our execution tests use partial client views.

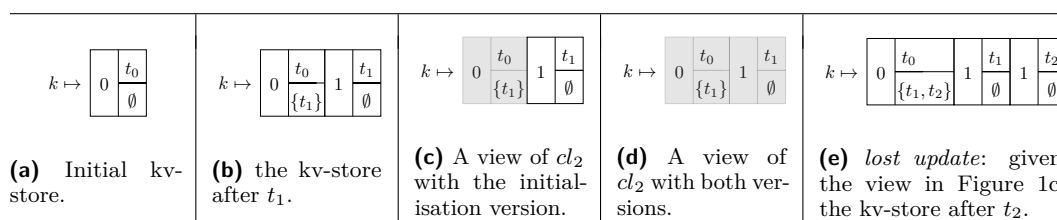
Nagar and Jagannathan [35] proposed a fine-grained interleaving operational semantics on abstract executions, and provide robustness results for client programs using a prototype model-checking tool. They do this by converting abstract executions to dependency graphs and checking the violation of robustness on the dependency graphs. We have two concerns with this approach. First, despite assuming atomic visibility of transactions, they present a fine-grained semantics at the level of the individual transactional operations rather than whole transactions, in order to capture *eventual consistency* [9]. In contrast, our semantics is coarse-grained in that the interleaving is at the level of whole transactions, and we instead capture *read atomic* [4], a variant of *eventual consistency* [9] for atomic transactions. Second, all the literature that performs client analysis on abstract executions [7, 12, 13, 14, 35], including the approach of Nagar and Jagannathan, achieves this indirectly by over-approximating the consistency-model specifications using dependency graphs. It is unknown how to do this precisely [14]. In contrast, we prove robustness results directly by analysing the structure of kv-stores, without over-approximation. We also give precise reasoning about the mutual exclusion of locks, which we believe will be difficult to prove using abstract executions.

Kaki et al. [27] proposed an operational semantics for SQL transactions over an abstract, centralised, single-version store, with consistency models given by the standard ANSI/SQL isolation levels [6]. They develop a program logic and prototype tool for reasoning about client programs, and so can capture invariant properties of the state. They can express SI, but they do not capture the weaker consistency models such as PSI which is an important consistency model for distributed databases. Kaki et al. have explored these weaker consistency models in follow-on work [26], but they focus on an axiomatic semantics for abstract executions over CRDTs not an operational semantics over kv-stores.

Finally, Koskinen and Parkinson [29] proposed a log-based semantics for verifying implementations that satisfy serialisability, based not only on kv-stores but also on other ADTs. Their work comprises a centralised global log and partial client-local logs, similar to our kv-stores and views. Their model focuses on serialisability. There is no evidence that it can be easily extended to tackle weaker consistency models.

2 Overview

We introduce our centralised operational semantics for describing the client-observable behaviours of atomic transactions updating distributed kv-stores. We show that our interleaving semantics provides an abstract interface for both verifying distributed protocols and proving invariant properties of client programs.



■ **Figure 1** Lost update anomaly: single counter.

Example. We use a simple transactional library, $\text{Counter}(k)$, to introduce our operational semantics. Clients of this library can manipulate the value of counter k via two transactional operations: $\text{Inc}(k) \triangleq [x := [k]; [k] := x+1]$ and $\text{Read}(k) \triangleq [x := [k]]$. The $x := [k]$ reads the value of k in local variable x ; and $[k] := x+1$ writes $x+1$ to k . The code of each operation is wrapped in square brackets, denoting a transaction that executes *atomically*.

Consider a replicated database where a client only interacts with one replica. For such a database, the behaviour of the atomic transactions is subtle, depending heavily on the particular consistency model under consideration. Consider the client program P_{LU} below:

$$P_{LU} \triangleq cl_1 : \text{Inc}(k) \parallel cl_2 : \text{Inc}(k)$$

where we assume that clients cl_1 and cl_2 work on different replicas and, for simplicity, each replica has a kv-store with just one key k . Initially, key k holds value 0 in all replicas. Intuitively, as transactions are executed atomically, after both calls to $\text{Inc}(k)$ have terminated, the counter should hold value 2. Indeed, this is the only outcome allowed under the *serialisability* (SER) consistency model, where transactions appear to execute in a sequential order, one after another. The implementation of SER in distributed kv-stores is known to come at a significant performance cost. Implementers are, therefore, content with *weaker* consistency models [3, 6, 8, 21, 32, 33, 38, 42]. For example, if replicas provide no synchronisation mechanism for transactions, it is possible for both clients to read the same initial value 0 for k at their distinct replicas, update it to 1, and eventually propagate their updates of k to other replicas. Thus, both replicas remain unchanged with value 1 for k . This weak behaviour is known as the *lost update* anomaly, which is allowed under *causal consistency* (CC), but not under *parallel snapshot isolation* (PSI) and *snapshot isolation* (SI).

Centralised Operational Semantics. Our operational semantics provides transitions over abstract states, comprising a centralised, multi-versioned *kv-store*, which is *global* in that it records all the versions written by all its clients, and a *client view*, which is *partial* in that it records only those versions in the kv-store observed by a client. Each transition of our operational semantics either updates a client-local variable stack using a primitive command, or updates the kv-store and client view using an atomic transaction. The atomic transactions are subject to an *execution test*, which analyses the state to determine if the associated update is allowed under the given consistency model.

We show how the lost update anomaly in P_{LU} is modelled in our operational semantics. A centralised kv-store provides an abstraction of the real-world replicated key-value store of our example. It is a function mapping keys to a *version* list, recording all the values written to the key together with information about the transactions that accessed it. The total order of versions on a key k is always known due to the resolution policy of the distributed database, for example last-write-wins. In the P_{LU} example, our initial centralised kv-store comprises a single key k with one initialisation version $(0, t_0, \emptyset)$. This version represents the initialisations

in both replicas where k holds value 0, the version *writer* is the initialising transaction t_0 (this version was written by t_0), and the version *reader set* is empty (no transaction has read this version). Figure 1a depicts this initial centralised kv-store, with the version represented as a box sub-divided in three sections: the value 0, the writer t_0 , and the reader set \emptyset .

Suppose that cl_1 first invokes $\text{Inc}(k)$ on Figure 1a. It does this by choosing a fresh transaction identifier t_1 , then reading the initial version of k with value 0 and writing a new value 1 for k . The resulting kv-store is depicted in Figure 1b, where the initial version of k has been updated to reflect that it has been read by t_1 and a new version with value 1 is installed at the end of the list. Now suppose that client cl_2 invokes $\text{Inc}(k)$ on Figure 1b. As there are now two versions available for k , we must determine the version from which cl_2 fetches its value. This is where the partial *client view* comes into play. Intuitively, a view of client cl_2 comprises those versions in the kv-store that are *visible* to cl_2 , i.e. those that can be read by cl_2 . If more than one version is visible, then the newest (right-most) version is selected, modelling the *last-write-wins* resolution policy used by many distributed key-value stores. In our example, there are two candidate views for cl_2 when running $\text{Inc}(k)$ on Figure 1b: one containing only the initial version of k as depicted in Figure 1c, and the other containing both versions of k as depicted in Figure 1d⁵. Given the cl_2 view in Figure 1c, client cl_2 chooses a fresh transaction identifier t_2 , reads the initial value 0 and writes a new version with value 1, as depicted in Figure 1e. Such a kv-store does not contain a version with value 2, despite two increments on k , producing the lost update anomaly. Had we used the the cl_2 view in Figure 1d instead, client cl_2 would have read the newest value 1 and written a new version with value 2.

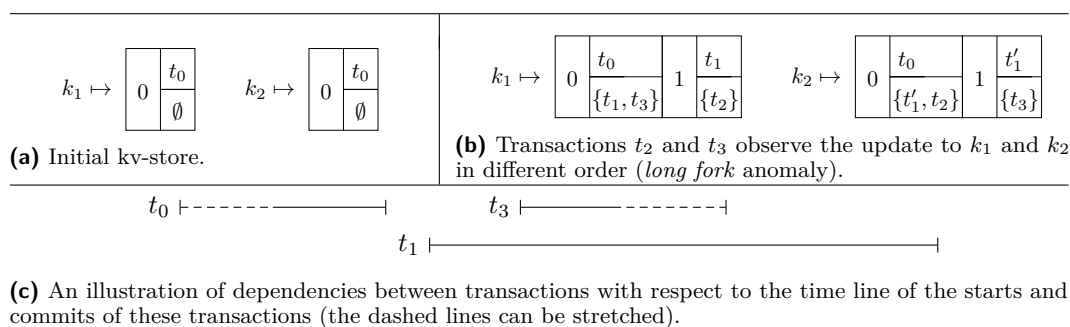
The lost update anomaly is allowed under the CC consistency model, and disallowed under SER, SI and PSI. To distinguish these cases, we use an *execution test* which directly restricts the updates that are possible at the point where the transaction commits. A simple way of doing this is to require that a client writing a transaction to k have a view containing *all* versions of k available in the global state. This prevents the situation where the view of cl_2 is that given in Figure 1c. This execution test corresponds to what is known in the literature as *write-conflict freedom* [11], which ensures that at most one concurrent transaction can write to a key at any one time.

The situation becomes more complicated when the library contains multiple counters where each client can read and increment several counters in one session. For instance, consider the following client program:

$$\begin{aligned} \text{P}_{\text{LF}} \triangleq & cl_1 : [\mathbf{x} := [k_1]; [k_1] := \mathbf{x} + 1]; [\mathbf{y} := [k_2]; [k_2] := \mathbf{y} + 1] \\ & || cl_2 : [\mathbf{x} := [k_1]; \mathbf{y} := [k_2]] || cl_3 : [\mathbf{x} := [k_1]; \mathbf{y} := [k_2]]. \end{aligned}$$

where, for simplicity, the kv-store has just the keys k_1 and k_2 (Figure 2a). Suppose that cl_1 executes both transactions first, writing 1 to k_1 and k_2 using fresh transaction identifiers t_1 and t'_1 , respectively. This results in k_1 and k_2 having two versions with values 0 and 1 each, as illustrated in Figure 2b. Client cl_2 next executes its transaction, identified by t_2 , using a view that contains both versions of k_1 but only the initial version of k_2 . This means that cl_2 reads 1 for k_1 and 0 for k_2 , i.e. cl_2 observes the increment of k_1 happening before that of k_2 . Symmetrically, cl_3 executes its transaction, identified by t_3 , using a view that contains both versions for k_2 but only the initial version of k_1 . As such, cl_3 reads 0 for k_1 and 1 for k_2 , i.e. cl_3 observes the increment of k_2 happening before that of k_1 . This behaviour is known as the *long fork* anomaly (Figure 2b).

⁵ As we explain in Section 3.1, we always require the client view to include the initial version of each key.



■ **Figure 2** Long fork anomaly: multiple counters.

The long fork anomaly is disallowed under strong models such as **SER** and **SI**, but is allowed under weaker models such as **PSI** and **CC**. To capture such consistency models and disallow the long fork anomaly of P_{LF} , we must strengthen the execution test associated with the kv-store. For **SER**, we simply strengthen the execution test by ensuring that a client can execute a transaction only if its view contains all versions available in the global state. For **SI**, the execution test is more subtle, requiring that a client view be a set of versions, i.e. *closed* with respect to the commit order of transactions. This means that if a client view includes a version written by a transaction t , then it must include all versions written by transactions that committed before t . Our kv-stores do not contain all the information about the commit order. However, we have enough information to determine the following commit order between transactions:

1. if a transaction, e.g. t_3 in Figure 2, reads a version written by another transaction, e.g. t_0 , then it must start after the commit of the transaction that wrote the version, e.g. t_3 must start after the commit of t_0 (Figure 2c);
2. if a transaction writes a newer version of a key, e.g. t_1 for k_1 , then it must commit after the transactions that wrote the previous versions of the key, e.g. t_0 (Figure 2c); and
3. if a transaction reads an older version of a key, e.g. t_3 for k_1 , it must start before the commit of all transactions that write the newer versions of k , e.g. t_1 (Figure 2c).

In Section 4, we formally define the execution tests associated with several consistency models on kv-stores and client views. In [46], we show the equivalence of our operational definitions of consistency models and the existing declarative definitions based on abstract executions [11], and hence those based on dependency graphs [1].

Verifying Implementation Protocols. The first application of our operational semantics is to show that implementation protocols of distributed key-value stores satisfy certain consistency models. We do this by representing the implementation protocol using our centralised operational semantics: our abstract states provide a faithful abstraction of replicated and partitioned databases, and our execution tests provide a faithful abstraction of the synchronisation mechanisms enforced by these databases when committing a transaction. We verify the correctness of our representation using trace refinement. Thus, a distributed protocol satisfies the particular consistency model associated with the particular execution test of our representation. We demonstrate that the COPS protocol [33] for implementing a replicated database satisfies our definition of **CC** (reported in Section 5.1 and proved in [46]), and the Clock-SI protocol [21] for implementing a partitioned database satisfies our definition of **SI** (given in [46]). Since our definitions of consistency model are equivalent

to those in the literature [46], we have demonstrated that COPS and Clock-SI satisfy the accepted general definitions of the respective consistency models. This contrasts with the previous results in [33] and [21] which demonstrated that these protocols satisfy specific consistency models defined for those particular implementations.

Proving Invariant Properties of Client Programs. The second application of our operational semantics is to prove invariant properties for transactional libraries (Section 5.2). One well-known property is *robustness*. A library is robust against a (weak) consistency model M if, for all its client programs P and all kv-stores \mathcal{K} , if \mathcal{K} is obtained by executing P under M , then \mathcal{K} can also be obtained under SER , i.e. library clients have no observable weak behaviours. We prove the robustness of the single counter library against PSI , and the robustness of a multi-counter library and the banking library of [2] against SI . We prove robustness against SI by proving general invariants that guarantee robustness against a new model we propose, WSI , which lies between PSI and SI . As we discuss in Section 5.2, although existing techniques [35, 12, 7] in the literature can verify such robustness properties, they typically do so by examining *full traces*. By contrast, we establish invariant properties at each execution step of our operational semantics, thus allowing a simpler, more compositional proof.

We also demonstrate the use of our operational semantics to prove library-specific invariant properties. In particular, we show that a lock library is correct against PSI , in that it satisfies the *mutual exclusion guarantee*, even though it is not robust against PSI . To do this, we encode this guarantee as an invariant of the lock library, establishing the invariant at each transition step of the operational semantics. By contrast, establishing such library-specific properties using the existing techniques is more difficult. This is because existing techniques [35, 12] do not directly record the library *state*; rather, they record full execution traces, making them less amenable for reasoning about such properties.

3 Operational Model

We define an interleaving operational semantics for atomic transactions (Section 3.2) on abstract states comprising global kv-stores and partial client views (Section 3.1). Our semantics is parametrised by an execution test which induces a consistency model (Section 4).

3.1 Abstract States: Key-Value Stores and Client Views

The abstract states of our operational semantics comprise a global, centralised kv-store and a partial client view. A kv-store comprises key-indexed lists of versions which record the history of the key with values and meta-data of the transactions that accessed it: the writer and readers.

We assume a countably infinite set of *client identifiers*⁶, $CLIENTID \ni cl$. The set of *transaction identifiers*, $TxID \ni t$, is defined by $TxID \triangleq \{t_0\} \uplus \{t_{cl}^n \mid cl \in CLIENTID \wedge n \geq 0\}$, where t_0 denotes the *initialisation transaction* and t_{cl}^n identifies a transaction committed by client cl with n determining the client session order: $SO \triangleq \{(t, t') \mid \exists cl, n, m. t = t_{cl}^n \wedge t' = t_{cl}^m \wedge n < m\}$. Subsets of $TxID$ are ranged over by T, T', \dots . We let $TxID_0 \triangleq TxID \setminus \{t_0\}$.

► **Definition 1 (Kv-stores).** Assume a countably infinite set of keys, $KEY \ni k$, and a countably infinite set of values, $VALUE \ni v$, which includes the keys and an initialisation value v_0 . The set of versions, $VERSION \ni \nu$, is $VERSION \triangleq VALUE \times TxID \times \mathcal{P}(TxID_0)$. A kv-store is a function $\mathcal{K} : KEY \rightarrow List(VERSION)$, where $List(VERSION) \ni \mathcal{V}$ is the set of lists of versions.

⁶ We use the notation $A \ni a$ to denote that elements of A are ranged over by a and its variants a', a_1, \dots .

Each version has the form $\nu=(v,t,T)$, where v is a value, the *writer* t identifies the transaction that wrote v , and the *reader set* T identifies the transactions that read v . We write $\text{val}(\nu)$, $\text{w}(\nu)$ and $\text{rs}(\nu)$ to project the components of ν . Given a kv-store \mathcal{K} and a transaction t , we write $t \in \mathcal{K}$ if t is either the writer or one of the readers of a version in \mathcal{K} ; we write $|\mathcal{K}(k)|$ for the length of the version list $\mathcal{K}(k)$, and $\mathcal{K}(k,i)$ for the i^{th} version of k in kv-store \mathcal{K} .

We assume that the version list of each key has an initialisation version carrying the initialisation value v_0 , written by the initialisation transaction t_0 with an initial empty reader set. We focus on kv-stores whose consistency model satisfies the *snapshot property*, ensuring that a transaction reads and writes at most one version for each key:

$$\forall k, i, j. (\text{rs}(\mathcal{K}(k,i)) \cap \text{rs}(\mathcal{K}(k,j)) \neq \emptyset \vee \text{w}(\mathcal{K}(k,i)) = \text{w}(\mathcal{K}(k,j))) \Rightarrow i = j \quad (\text{snapshot})$$

This is a standard assumption for distributed databases, e.g. in [3, 4, 6, 8, 21, 33, 38, 42]. Finally, we assume that the kv-store agrees with the session order of clients, in that a client cannot read a version of a key that has been written by a future transaction within the same session, and the order in which versions are written by a client must agree with its session order, i.e. for any k, i, j, t, t' :

$$t = \text{w}(\mathcal{K}(k,i)) \wedge t' \in \text{rs}(\mathcal{K}(k,i)) \Rightarrow (t', t) \notin \text{SO}^? \quad (\text{wr-so})$$

$$t = \text{w}(\mathcal{K}(k,i)) \wedge t' = \text{w}(\mathcal{K}(k,j)) \wedge i < j \Rightarrow (t', t) \notin \text{SO}^? \quad (\text{ww-so})$$

A kv-store is *well-formed* if it satisfies these assumptions. Henceforth, we assume kv-stores are well-formed, and let KVS denote the set of well-formed kv-stores.

A global kv-store provides an abstract centralised description of updates associated with distributed kv-stores that is *complete* in that no update has been lost in the description. By contrast, in both replicated and partitioned distributed databases, a client may have incomplete information about updates distributed between machines. We model this incomplete information by defining a *client view*, or just *view*, of the kv-store which provides a *partial* record of the updates observed by a client. We require that a client view be *atomic* in that it can see either all or none of the updates of a transaction. This client view was partly inspired by the views of the “promising” C11 operational semantics [28].

► **Definition 2 (Views).** A view of a kv-store $\mathcal{K} \in \text{KVS}$ is a function $u \in \text{VIEWS}(\mathcal{K}) \triangleq \text{KEY} \rightarrow \mathcal{P}(\mathbb{N})$ such that, for all i, i', k, k' :

$$0 \in u(k) \wedge (i \in u(k) \Rightarrow 0 \leq i < |\mathcal{K}(k)|) \quad (\text{in-range})$$

$$i \in u(k) \wedge \text{w}(\mathcal{K}(k,i)) = \text{w}(\mathcal{K}(k',i')) \Rightarrow i' \in u(k') \quad (\text{atomic})$$

Given two views $u, u' \in \text{VIEWS}(\mathcal{K})$, the order between them is defined by $u \sqsubseteq u' \stackrel{\text{def}}{\Leftrightarrow} \forall k \in \text{dom}(\mathcal{K}). u(k) \subseteq u'(k)$. The set of views is $\text{VIEWS} \triangleq \bigcup_{\mathcal{K} \in \text{KVS}} \text{VIEWS}(\mathcal{K})$. The initial view, u_0 , is defined by $u_0(k) = \{0\}$ for every $k \in \text{KEY}$.

Our operational semantics updates *configurations*, which are pairs comprising a kv-store and a function describing the views of a finite set of clients.

► **Definition 3 (Configurations).** A configuration, $\Gamma \in \text{CONF}$, is a pair $(\mathcal{K}, \mathcal{U})$ with $\mathcal{K} \in \text{KVS}$ and $\mathcal{U} : \text{CLIENTID} \xrightarrow{\text{fin}} \text{VIEWS}(\mathcal{K})$. The set of initial configurations, $\text{CONF}_0 \subseteq \text{CONF}$, contains configurations of the form $(\mathcal{K}_0, \mathcal{U}_0)$, where \mathcal{K}_0 is the initial kv-store defined by $\mathcal{K}_0(k) \triangleq (v_0, t_0, \emptyset)$ for all $k \in \text{KEY}$.

Given a configuration $(\mathcal{K}, \mathcal{U})$ and a client cl , if $u = \mathcal{U}(cl)$ is defined then, for each k , the configuration determines the sub-list of versions in \mathcal{K} that cl sees. If $i, j \in u(k)$ and $i < j$, then cl sees the values carried by versions $\mathcal{K}(k, i)$ and $\mathcal{K}(k, j)$, and it also sees that the version $\mathcal{K}(k, j)$ is more up-to-date than $\mathcal{K}(k, i)$. It is therefore possible to associate a *snapshot* with the view u , which identifies, for each key k , the last version included in the view. This definition assumes that the database satisfies the *last-write-wins* resolution policy, employed by many distributed key-value stores. However, our formalism can be adapted straightforwardly to capture other resolution policies.

► **Definition 4** (View Snapshots). *Given $\mathcal{K} \in \text{KVS}$ and $u \in \text{VIEWS}(\mathcal{K})$, the view snapshot of u in \mathcal{K} is a function, $\text{snapshot}(\mathcal{K}, u) : \text{KEY} \rightarrow \text{VALUE}$, defined by:*

$$\text{snapshot}(\mathcal{K}, u) \triangleq \lambda k. \text{val}(\mathcal{K}(k, \max_{<}(u(k))))$$

where $\max_{<}(u(k))$ is the maximum element in $u(k)$ under the natural order $<$ on \mathbb{N} .

When clear from the context, we simply refer to a view snapshot as a *snapshot*.

3.2 Operational Semantics

Core Programming Language. We assume a language of expressions built from values v and program variables \mathbf{x} , defined by: $\mathbf{E} ::= v \mid \mathbf{x} \mid \mathbf{E} + \mathbf{E} \mid \dots$. The *evaluation* $\llbracket \mathbf{E} \rrbracket_s$ of expression \mathbf{E} is parametric in the client-local stack s : $\llbracket v \rrbracket_s \triangleq v$ $\llbracket \mathbf{x} \rrbracket_s \triangleq s(\mathbf{x})$ $\llbracket \mathbf{E}_1 + \mathbf{E}_2 \rrbracket_s \triangleq \llbracket \mathbf{E}_1 \rrbracket_s + \llbracket \mathbf{E}_2 \rrbracket_s$ \dots . A *program* \mathbf{P} comprises a finite number of clients, where each client is associated with a unique identifier $cl \in \text{CLIENTID}$, and executes a sequential *command* \mathbf{C} , defined by:

$$\begin{aligned} \mathbf{C} &::= \text{skip} \mid \mathbf{C}_p \mid [\mathbf{T}] \mid \mathbf{C} ; \mathbf{C} \mid \mathbf{C} + \mathbf{C} \mid \mathbf{C}^* & \mathbf{C}_p &::= \mathbf{x} := \mathbf{E} \mid \text{assume}(\mathbf{E}) \\ \mathbf{T} &::= \text{skip} \mid \mathbf{T}_p \mid \mathbf{T} ; \mathbf{T} \mid \mathbf{T} + \mathbf{T} \mid \mathbf{T}^* & \mathbf{T}_p &::= \mathbf{C}_p \mid \mathbf{x} := [\mathbf{E}] \mid [\mathbf{E}] := \mathbf{E} \end{aligned}$$

Sequential commands (\mathbf{C}) comprise **skip**, primitive commands (\mathbf{C}_p), atomic transactions ($[\mathbf{T}]$), and standard compound constructs: sequential composition ($;$), non-deterministic choice ($+$) and iteration ($*$). Primitive commands include variable assignment ($\mathbf{x} := \mathbf{E}$) and assume statements (**assume**(\mathbf{E})) which can be used to encode conditionals. They are used for computations based on client-local variables and can hence be invoked without restriction. Transactional commands (\mathbf{T}) comprises **skip**, primitive transactional commands (\mathbf{T}_p), and the standard compound constructs. Primitive transactional commands comprise primitive commands as well as lookup ($\mathbf{x} := [\mathbf{E}]$) and mutation ($[\mathbf{E}] := \mathbf{E}$) used, respectively, to read and write a single key to a kv-store, and can only be invoked within an atomic transaction.

A *program* \mathbf{P} is a finite partial function from client identifiers to sequential commands. For clarity, we often write $\mathbf{C}_1 \parallel \dots \parallel \mathbf{C}_n$ for a program with n clients identified by $cl_1 \dots cl_n$, with each client cl_i executing \mathbf{C}_i . Each client cl_i is associated with a client-local *stack*, $s_i \in \text{STACK} \triangleq \text{VAR} \rightarrow \text{VALUE}$, mapping program variables (ranged over by $\mathbf{x}, \mathbf{y}, \dots$) to values.

Transactional Semantics. In our operational semantics, transactions are executed *atomically*. It is still possible for an implementation, e.g. COPS [33], to update the underlying distributed kv-stores while the transaction is in progress. It just means that, given the abstractions captured by our global kv-stores and partial client views, such an update is modelled as an instantaneous atomic update. Intuitively, given a configuration $\Gamma = (\mathcal{K}, \mathcal{U})$, when a client cl executes a transaction $[\mathbf{T}]$, it performs the following steps:

TPRIMITIVE $\frac{(s, \sigma) \xrightarrow{\text{T}_p} (s', \sigma') \quad o = \text{op}(s, \sigma, \text{T}_p)}{(s, \sigma, \mathcal{F}), \text{T}_p \rightsquigarrow (s', \sigma', \mathcal{F} \ll o), \text{skip}}$	$\mathcal{F} \ll (\text{R}, k, v) \triangleq \begin{cases} \mathcal{F} \cup \{(\text{R}, k, v)\} & \text{if } \forall l, v'. (l, k, v') \notin \mathcal{F} \\ \mathcal{F} & \text{otherwise} \end{cases}$ $\mathcal{F} \ll (\text{W}, k, v) \triangleq (\mathcal{F} \setminus \{(\text{W}, k, v') \mid v' \in \text{VALUE}\}) \cup \{(\text{W}, k, v)\}$ $\mathcal{F} \ll \epsilon \triangleq \mathcal{F}$
$(s, \sigma) \xrightarrow{x := \text{E}} (s[x \mapsto \llbracket \text{E} \rrbracket_s], \sigma) \quad (s, \sigma) \xrightarrow{\text{assume}(\text{E})} (s, \sigma) \text{ where } \llbracket \text{E} \rrbracket_s \neq 0$ $(s, \sigma) \xrightarrow{x := \llbracket \text{E} \rrbracket} (s[x \mapsto \sigma(\llbracket \text{E} \rrbracket_s)], \sigma) \quad (s, \sigma) \xrightarrow{\llbracket \text{E}_1 \rrbracket := \text{E}_2} (s, \sigma[\llbracket \text{E}_1 \rrbracket_s \mapsto \llbracket \text{E}_2 \rrbracket_s])$	
$\text{op}(s, \sigma, x := \text{E}) \triangleq \epsilon \quad \text{op}(s, \sigma, \text{assume}(\text{E})) \triangleq \epsilon$ $\text{op}(s, \sigma, x := \llbracket \text{E} \rrbracket) \triangleq (\text{R}, \llbracket \text{E} \rrbracket_s, \sigma(\llbracket \text{E} \rrbracket_s)) \quad \text{op}(s, \sigma, \llbracket \text{E}_1 \rrbracket := \text{E}_2) \triangleq (\text{W}, \llbracket \text{E}_1 \rrbracket_s, \llbracket \text{E}_2 \rrbracket_s)$	

■ **Figure 3** The semantics of transactional commands.

1. it constructs an initial *snapshot* σ of \mathcal{K} using its view $\mathcal{U}(cl)$ as described in Definition 4;
2. it executes T in isolation over σ accumulating the effects (the reads and writes) of executing T; and
3. it commits T by incorporating these effects into \mathcal{K} .

► **Definition 5** (Transactional snapshots). *A transactional snapshot, $\sigma \in \text{SNAPSHOT} \triangleq \text{KEY} \rightarrow \text{VALUE}$, is a function from keys to values.*

When clear from the context, we simply refer to a transactional snapshot as a *snapshot*.

The rules for transactional commands (Figure 3) are defined using an arbitrary transactional snapshot. The rules for sequential commands and programs (Figure 4) are defined using a transactional snapshot given by a view snapshot. To capture the effects of executing a transaction T on a snapshot σ of kv-store \mathcal{K} , we identify a *fingerprint* of T on σ which captures the first values T reads from σ , and the last values T writes to σ and intends to commit to \mathcal{K} . Execution of a transaction in a given configuration and variable stack may result in more than one fingerprint due to non-determinism (non-deterministic choice).

► **Definition 6** (Fingerprints). *Let OP denote the set of read (R) and write (W) operations defined by $OP \triangleq \{(l, k, v) \mid l \in \{\text{R}, \text{W}\} \wedge k \in \text{KEY} \wedge v \in \text{VALUE}\}$. A fingerprint \mathcal{F} is a set of operations, $\mathcal{F} \subseteq OP$, such that: $\forall k \in \text{KEY}, l \in \{\text{R}, \text{W}\}. (l, k, v_1), (l, k, v_2) \in \mathcal{F} \Rightarrow v_1 = v_2$.*

A fingerprint contains at most one read operation and at most one write operation for a given key. This reflects our assumption regarding transactions that satisfy the snapshot property: reads are taken from a single snapshot of the kv-store; and only the last write of a transaction to each key is committed to the kv-store.

The rule for primitive transactional commands, TPRIMITIVE, is given in Figure 3. The rules for the compound constructs are straightforward and given in [46]. The TPRIMITIVE rule updates the snapshot and the fingerprint of a transaction: the premise $(s, \sigma) \xrightarrow{\text{T}_p} (s', \sigma')$ describes how executing T_p affects the local state (the client stack and the snapshot) of a transaction; and the premise $o = \text{op}(s, \sigma, \text{T}_p)$ identifies the operation on the kv-store associated with T_p , where the empty operation ϵ is used for those primitive commands that do not contribute to the fingerprint.

The conclusion of TPRIMITIVE uses the *combination operator* $\ll : \mathcal{P}(OP) \times (OP \uplus \{\epsilon\}) \rightarrow \mathcal{P}(OP)$, defined in Figure 3, to extend the fingerprint \mathcal{F} accumulated with operation o associated with T_p , as appropriate: it adds a read from k if \mathcal{F} contains no entry for k , and it always updates the write for k to \mathcal{F} , removing previous writes to k .

$$\begin{array}{c}
\text{CPRIMITIVE} \\
\frac{s \xrightarrow{C_p} s'}{cl \vdash (\mathcal{K}, u, s), C_p \xrightarrow{(cl, \iota)}_{\text{ET}} (\mathcal{K}, u, s'), \text{skip}} \quad s \xrightarrow{x := E} s [x \mapsto \llbracket E \rrbracket_s] \\
s \xrightarrow{\text{assume}(E)} s \text{ where } \llbracket E \rrbracket_s \neq 0 \\
\text{CATOMICTRANS} \\
\frac{u \sqsubseteq u'' \quad \sigma = \text{snapshot}(\mathcal{K}, u'') \quad (s, \sigma, \emptyset), T \rightsquigarrow^* (s', _, \mathcal{F}), \text{skip} \quad \text{canCommit}_{\text{ET}}(\mathcal{K}, u'', \mathcal{F}) \\
t \in \text{NextTxID}(cl, \mathcal{K}) \quad \mathcal{K}' = \text{UpdateKV}(\mathcal{K}, u'', \mathcal{F}, t) \quad \text{vShift}_{\text{ET}}(\mathcal{K}, u'', \mathcal{K}', u')}{cl \vdash (\mathcal{K}, u, s), [T] \xrightarrow{(cl, u'', \mathcal{F})}_{\text{ET}} (\mathcal{K}', u', s'), \text{skip}} \\
\text{PPROG} \\
\frac{u = \mathcal{U}(cl) \quad s = \mathcal{E}(cl) \quad \mathcal{C} = \mathcal{P}(cl) \quad cl \vdash (\mathcal{K}, u, s), \mathcal{C} \xrightarrow{\lambda}_{\text{ET}} (\mathcal{K}', u', s'), \mathcal{C}'}{\vdash (\mathcal{K}, \mathcal{U}, \mathcal{E}), \mathcal{P} \xrightarrow{\lambda}_{\text{ET}} (\mathcal{K}', \mathcal{U}[cl \mapsto u'], \mathcal{E}[cl \mapsto s']), \mathcal{P}[cl \mapsto \mathcal{C}']}
\end{array}$$

■ **Figure 4** The semantics of sequential commands and programs.

Command and Program Semantics. We give the operational semantics of commands and programs in Figure 4. The command semantics describes transitions of the form $cl \vdash (\mathcal{K}, u, s), \mathcal{C} \xrightarrow{\lambda}_{\text{ET}} (\mathcal{K}', u', s'), \mathcal{C}'$ stating that, given the kv-store \mathcal{K} , client view u and stack s , a client cl may execute command \mathcal{C} for one step, updating the kv-store to \mathcal{K}' , the stack to s' , the view to u' and the command to its continuation \mathcal{C}' . The label λ is either of the form (cl, ι) denoting that cl executed a primitive command that required no access to \mathcal{K} , or (cl, u'', \mathcal{F}) denoting that cl committed an atomic transaction with final fingerprint \mathcal{F} under the view u'' . The semantics is parametric in the choice of the *execution test* ET , which is used to generate the *consistency model* under which a transaction can execute. In Section 4, we give several examples of execution tests for well-known consistency models. In [46], we prove that the consistency models generated by our execution tests are equivalent to their corresponding existing definitions using abstract executions.

The rules for compound constructs are straightforward and given in [46]. The rule for primitive commands, CPRIMITIVE , depends on the transition system $\rightsquigarrow \subseteq \text{STACK} \times \text{STACK}$ which describes how the primitive command C_p affects the stack. The CATOMICTRANS rule describes the execution of an atomic transaction under the execution test ET .

We explain the CATOMICTRANS rule in detail. The first premise states that the current view u of the executing command may be advanced to a newer view u'' (see Definition 2). Given the new view u'' , the transaction obtains a snapshot σ of the kv-store \mathcal{K} , and executes T locally to completion (skip), updating the stack to s' , while accumulating the fingerprint \mathcal{F} , as described by the second and third premises of CATOMICTRANS . Note that the resulting snapshot is ignored as the effect of the transaction is recorded in the fingerprint \mathcal{F} . The $\text{canCommit}_{\text{ET}}(\mathcal{K}, u'', \mathcal{F})$ premise ensures that, under the execution test ET , the final fingerprint \mathcal{F} of the transaction is compatible with the (original) kv-store \mathcal{K} and the client view u'' , and thus the transaction *can commit*. Observe that the canCommit check is parametric in the execution test ET . This is because the conditions checked upon committing depend on the consistency model under which the transaction is to commit. In Section 4, we define canCommit for several execution tests associated with well-known consistency models.

Client cl is now ready to commit the transaction resulting in the kv-store \mathcal{K}' with the client view u'' *shifting* to a new view u' and proceeds as follows:

1. it picks a fresh transaction identifier $t \in \text{NextTxID}(cl, \mathcal{K})$;
2. computes the new kv-store $\mathcal{K}' = \text{UpdateKV}(\mathcal{K}, u'', \mathcal{F}, t)$; and
3. checks if the *view shift* is permitted under ET using $\text{vShift}_{\text{ET}}(\mathcal{K}, u'', \mathcal{K}', u')$.

Note that as with `canCommit`, the `vShift` check is parametric in the execution test `ET`. This is because the conditions checked for shifting the client view depend on the consistency model. In Section 4 we define `vShift` for several execution tests associated with well-known consistency models. The set `NextTxID`(cl, \mathcal{K}) is given by: $\{t_{cl}^m \mid \forall m. t_{cl}^m \in \mathcal{K} \Rightarrow m < n\}$. The function `UpdateKV`($\mathcal{K}, u, \mathcal{F}, t$) describes how the fingerprint \mathcal{F} of transaction t executed under view u updates kv-store \mathcal{K} : for each read $(R, k, v) \in \mathcal{F}$, it adds t to the reader set of the last version of k in u ; for each write (W, k, v) , it appends a new version (v, t, \emptyset) to $\mathcal{K}(k)$. The function `UpdateKV` is well-formed, because a fingerprint contains at most one write operation and one read operation for a given key (see [46] for the full details).

► **Definition 7** (Transactional update). *The function `UpdateKV`($\mathcal{K}, u, \mathcal{F}, t$) is defined as:*

$$\begin{aligned} \text{UpdateKV}(\mathcal{K}, u, \emptyset, t) &\triangleq \mathcal{K} \\ \text{UpdateKV}(\mathcal{K}, u, \{(R, k, v)\} \uplus \mathcal{F}, t) &\triangleq \text{let } i = \max_{<}(u(k)) \text{ and } (v, t', T) = \mathcal{K}(k, i) \text{ in} \\ &\quad \text{UpdateKV}(\mathcal{K}[k \mapsto \mathcal{K}(k)[i \mapsto (v, t', T \uplus \{t\})]], u, \mathcal{F}, t) \\ \text{UpdateKV}(\mathcal{K}, u, \{(W, k, v)\} \uplus \mathcal{F}, t) &\triangleq \text{let } \mathcal{K}' = \mathcal{K}[k \mapsto \mathcal{K}(k) :: (v, t, \emptyset)] \text{ in } \text{UpdateKV}(\mathcal{K}', u, \mathcal{F}, t) \end{aligned}$$

where $\mathcal{V}[i \mapsto \nu] \triangleq \nu_0 :: \dots :: \nu_{i-1} :: \nu :: \nu_{i+1} :: \dots :: \nu_n$ for all version lists $\mathcal{V} = \nu_0 :: \dots :: \nu_n$ and indexes $i : 0 \leq i \leq n$.

The last rule, `PPROG` (Figure 4), captures the execution of a program step using a *client environment*, $\mathcal{E} \in \text{CENV}$, which is a function from client identifiers to stacks associating each client with its stack. We assume that the domain of a client environment contains the domain of the program throughout the execution: $\text{dom}(\mathcal{P}) \subseteq \text{dom}(\mathcal{E})$. Program transitions are simply defined in terms of the transitions of their constituent client commands. This yields an interleaving semantics for transactions of different clients: a client executes a transaction in an atomic step without interference from the other clients.

4 Consistency Models Using Execution Tests on Kv-stores

We define what it means for a kv-store to be in a consistent state. Many different consistency models for distributed databases have been proposed in the literature, e.g. [3, 6, 8, 21, 32, 33, 38, 42], which capture different trade-offs between performance and application correctness. Example consistency models range from *serialisability*, a strong model which only allows kv-stores obtained from a serial execution of transactions with inevitable performance drawbacks, to *eventual consistency*, a weak model which imposes few conditions on the structure of kv-stores, leading to good performance but anomalous behaviours. We define consistency models for our kv-stores, by introducing the notion of an *execution test*, specifying whether a client is allowed to commit a transaction in a given kv-store. An execution test `ET` induces a consistency model as the set of kv-stores obtained by having clients non-deterministically commit transactions, so long as the constraints imposed by `ET` are satisfied. We explore a range of execution tests associated with well-known consistency models in the literature. In [46], we demonstrate that our operational definitions of consistency models over kv-stores using execution tests are equivalent to the established declarative definitions of consistency models over abstract executions [9, 11].

► **Definition 8** (Execution tests). *An execution test, `ET`, is a set of tuples, $\text{ET} \subseteq \text{KVS} \times \text{VIEWS} \times \text{FP} \times \text{KVS} \times \text{VIEWS}$, such that for all $(\mathcal{K}, u, \mathcal{F}, \mathcal{K}', u') \in \text{ET}$:*

1. $u \in \text{VIEWS}(\mathcal{K})$ and $u' \in \text{VIEWS}(\mathcal{K}')$;
2. `canCommit`_{ET}($\mathcal{K}, u, \mathcal{F}$);
3. `vShift`_{ET}($\mathcal{K}, u, \mathcal{K}', u'$); and
4. for all $k \in \mathcal{K}$ and $v \in \text{VALUE}$, if $(R, k, v) \in \mathcal{F}$ then $\mathcal{K}(k, \max_{<}(u(k))) = v$.

Intuitively, $(\mathcal{K}, u, \mathcal{F}, \mathcal{K}', u') \in \text{ET}$ means that, under the execution test ET, a client with initial view u over kv-store \mathcal{K} can commit a transaction with fingerprint \mathcal{F} to obtain the resulting kv-store \mathcal{K}' (given by Definition 7) while shifting its view to u' . Note that the last condition in Definition 8 enforces the last-write-wins policy [45]: a transaction always reads the most recent writes from the initial view u .

► **Definition 9** (Consistency models). *The consistency model induced by an execution test ET is defined as: $\text{CM}(\text{ET}) \triangleq \{\mathcal{K} \mid \exists \mathcal{K}_0, \mathcal{U}_0, \mathcal{E}, \mathcal{P}. (\mathcal{K}_0, \mathcal{U}_0, \mathcal{E}), \mathcal{P} \xrightarrow{\text{ET}}^* (\mathcal{K}, _, _, _)\}$.*

The largest execution test is denoted by ET_\top , where for all $\mathcal{K}, \mathcal{K}', u, u', \mathcal{F}$:

$$\text{canCommit}_{\text{ET}_\top}(\mathcal{K}, u, \mathcal{F}) \stackrel{\text{def}}{\iff} \text{true} \quad \text{and} \quad \text{vShift}_{\text{ET}_\top}(\mathcal{K}, u, \mathcal{K}', u') \stackrel{\text{def}}{\iff} \text{true}$$

The consistency model induced by ET_\top corresponds to the *Read Atomic* model [4], a variant of *Eventual Consistency* [9] for atomic transactions.

We present several examples of execution tests which give rise to consistency models on kv-stores. Recall that the snapshot property and the last-write-wins policy are hard-wired in our framework. As such, we can only define consistency models that satisfy these two constraints. Although this prohibits interesting consistency models such as *Read Committed*, we can express a large number of consistency models employed by distributed kv-stores.

Notation. Given relations $r, r' \subseteq A \times A$, we write: $r^?$, r^+ and r^* for the reflexive, transitive and reflexive-transitive closures of r , respectively; r^{-1} for the inverse of r ; $a_1 \xrightarrow{r} a_2$ for $(a_1, a_2) \in r$; and $r; r'$ for $\{(a_1, a_2) \mid \exists a. (a_1, a) \in r \wedge (a, a_2) \in r'\}$.

Recall that an execution test ET is a tuple $(\mathcal{K}, u, \mathcal{F}, \mathcal{K}', u')$ such that $\text{canCommit}_{\text{ET}}(\mathcal{K}, u, \mathcal{F})$ and $\text{vShift}_{\text{ET}}(\mathcal{K}, u, \mathcal{K}', u')$ hold (Definition 8). We proceed with several auxiliary definitions that allow us to define canCommit and vShift for several consistency models.

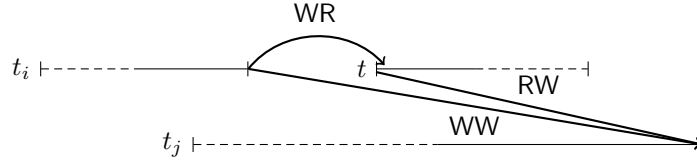
Prefix Closure. The set of visible transactions of a kv-store \mathcal{K} and a view u is: $\text{visTx}(\mathcal{K}, u) \triangleq \{\mathbf{w}(\mathcal{K}(k, i)) \mid i \in u(k)\}$. Given a relation on transactions, $R \subseteq \text{TxID} \times \text{TxID}$, a view u is closed with respect to a kv-store \mathcal{K} and R , written $\text{closed}(\mathcal{K}, u, R)$, if and only if:

$$\text{visTx}(\mathcal{K}, u) = ((R^*)^{-1}(\text{visTx}(\mathcal{K}, u))) \setminus \{t \mid \forall k \in \mathcal{K}, i. t \neq \mathbf{w}(\mathcal{K}(k, i))\}$$

That is, if transaction t is visible in u ($t \in \text{visTx}(\mathcal{K}, u)$), then all transactions t' that are R^* -before t ($t' \in (R^*)^{-1}(t)$) and are not read-only $t' \notin \{t'' \mid \forall k, i. t'' \neq \mathbf{w}(\mathcal{K}(k, i))\}$ are also visible in u ($t' \in \text{visTx}(\mathcal{K}, u)$).

Dependency Relations. We next define transactional dependency relations for kv-stores. Figure 7a illustrates an example kv-store and its transactional dependency relations. Given a kv-store \mathcal{K} , a key k and indexes i, j such that $0 \leq i < j < |\mathcal{K}(k)|$, if there exists t_i, T_i, t such that $\mathcal{K}(k, i) = (_, t_i, T_i)$, $\mathcal{K}(k, j) = (_, t_j, _)$ and $t \in T_i$, then for every key k :

1. there is a *Write-Read* dependency from t_i to t , written $(t_i, t) \in \text{WR}_{\mathcal{K}}(k)$, which intuitively means that t_i commits before t starts, as depicted in Figure 5;
2. there is a *Write-Write* dependency from t_i to t_j , written $(t_i, t_j) \in \text{WW}_{\mathcal{K}}(k)$, which intuitively means that t_i commits before t_j commits, as depicted in Figure 5; and
3. if $t \neq t_j$, then there is a *Read-Write* anti-dependency from t to t_j , written $(t, t_j) \in \text{RW}_{\mathcal{K}}(k)$, which intuitively means that t starts before t_j commits, as depicted in Figure 5.



■ **Figure 5** An example of dependencies between transactions with respect to the time line of the starts and commits of these transactions (dashed line being able to stretched).

ET	$\text{canCommit}_{\text{ET}}(\mathcal{K}, u, \mathcal{F}) \triangleq \text{closed}(\mathcal{K}, u, R_{\text{ET}})$	$\text{vShift}_{\text{ET}}(\mathcal{K}, u, \mathcal{K}', u')$
MR	true	$u \sqsubseteq u'$
RYW	true	$\forall t \in \mathcal{K}' \setminus \mathcal{K}. \forall k, i. (\text{w}(\mathcal{K}'(k, i), t) \in \text{SO}^? \Rightarrow i \in u'(k))$
CC	$R_{\text{CC}} \triangleq \text{SO} \cup \text{WR}_{\mathcal{K}}$	$\text{vShift}_{\text{MR} \cap \text{RYW}}(\mathcal{K}, u, \mathcal{K}', u')$
UA	$R_{\text{UA}} \triangleq \bigcup_{(w, k, _) \in \mathcal{F}} \text{WW}_{\mathcal{K}}^{-1}(k)$	true
PSI	$R_{\text{PSI}} \triangleq R_{\text{UA}} \cup R_{\text{CC}} \cup \text{WW}_{\mathcal{K}}$	$\text{vShift}_{\text{MR} \cap \text{RYW}}(\mathcal{K}, u, \mathcal{K}', u')$
CP	$R_{\text{CP}} \triangleq \text{SO}; \text{RW}_{\mathcal{K}}^? \cup \text{WR}_{\mathcal{K}}; \text{RW}_{\mathcal{K}}^? \cup \text{WW}_{\mathcal{K}}$	$\text{vShift}_{\text{MR} \cap \text{RYW}}(\mathcal{K}, u, \mathcal{K}', u')$
SI	$R_{\text{SI}} \triangleq R_{\text{UA}} \cup R_{\text{CP}} \cup (\text{WW}_{\mathcal{K}}; \text{RW}_{\mathcal{K}})$	$\text{vShift}_{\text{MR} \cap \text{RYW}}(\mathcal{K}, u, \mathcal{K}', u')$
SER	$R_{\text{SER}} \triangleq \text{WW}_{\mathcal{K}}^{-1}$	true

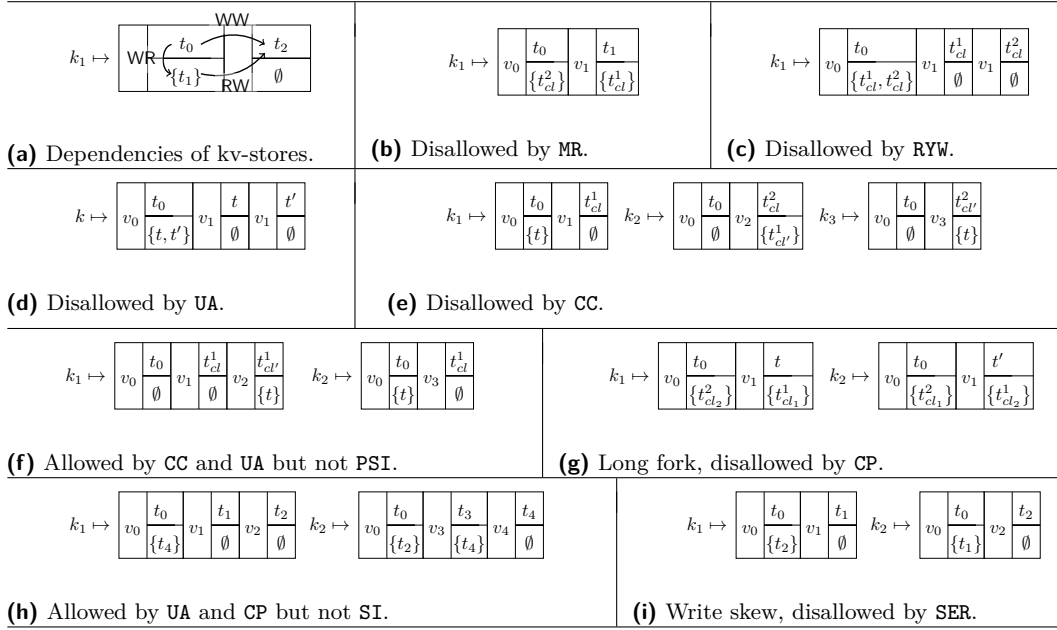
■ **Figure 6** Execution tests of consistency models defined by `canCommit` and `vShift` predicates, where `SO` is as given in Section 3.1.

In centralised databases, where there is a global notion of time, these dependency relations can be determined by the start and commit time of transaction as in Figure 5. However, in general, there is no global notion of time in distributed databases. In such settings, the write-read dependency `WR` is induced when a transaction reads from another transaction; the write-write dependency `WW` is given by the *last-write-wins* resolution policy, ordering the transactions that write to the same key; and the read-write anti-dependency `RW` is derived from `WR` and `WW`: if $(t, t') \in \text{WR}$ and $(t, t'') \in \text{WW}$, then $(t', t'') \in \text{RW}$. We adopt the same names as the dependency relations of dependency graphs [1] to underline the similarity. However, our relations here do *not* depend on those relations in dependency graphs.

We give several definitions of execution tests using `vShift` and `canCommit` in Figure 6.

Monotonic Reads (MR). This consistency model states that, when committing, a client cannot lose information in that it can only see increasingly more up-to-date versions from a kv-store. This prevents, for example, the kv-store of Figure 7b, since client cl first reads the latest version of k in t_{cl}^1 , and then reads the older, initial version of k in t_{cl}^2 . As such, the $\text{vShift}_{\text{MR}}$ predicate in Figure 6 ensures that clients can only extend their views. When this is the case, clients can *always* commit their transactions, and thus $\text{canCommit}_{\text{MR}}$ is simply **true**.

Read Your Writes (RYW). This consistency model states that a client must always see all the versions written by the client itself. The $\text{vShift}_{\text{RYW}}$ predicate thus states that after executing a transaction, a client contains all the versions it wrote in its view. This ensures that such versions will be included in the view of the client when committing future transactions. Note that under `RYW` the kv-store in Figure 7c is prohibited as the initial version of k holds value v_0 and client cl tries to update the value of k twice. For its first transaction t_{cl}^1 , it reads the initial value v_0 and then writes a new version with value v_1 . For its second transaction t_{cl}^2 , it reads the initial value v_0 again and writes a new version with value v_1 . The $\text{vShift}_{\text{RYW}}$ predicate rules out this example by requiring the client view after committing t_{cl}^1 to include the version it wrote. When this is the case, clients can always commit their transactions, and thus $\text{canCommit}_{\text{RYW}}$ is simply **true**.



■ **Figure 7** Behaviours disallowed under different consistency models. Figure 7a shows the dependencies of transactions in kv-stores (values omitted).

The MR and RYW models, together with the *monotonic writes* (MW) and *write follows reads* (WFR) models, are collectively known as *session guarantees*. Due to space constraints, the definitions associated with MW and WFR are given in [46].

We now give the definitions of well-known consistency models in distributed databases, including CC [9, 33, 40], PSI [3, 42], SI [6] and SER [37]. The $\text{vShift}_{\text{MR} \cap \text{RYW}}(\mathcal{K}, u, \mathcal{K}', u')$ = $\text{vShift}_{\text{MR}}(\mathcal{K}, u, \mathcal{K}', u') \cap \text{vShift}_{\text{RYW}}(\mathcal{K}, u, \mathcal{K}', u')$. The $\text{canCommit}_{\text{ET}}(\mathcal{K}, u, \mathcal{F})$ relation is defined by $\text{canCommit}_{\text{ET}}(\mathcal{K}, u, \mathcal{F}) \triangleq \text{closed}(\mathcal{K}, u, R_{\text{ET}})$ where R_{ET} is given for each execution test in Figure 6 as a combination of SO and the dependency relations. We use two less-known consistency models, *update atomic* (UA) and *consistent prefix* (CP). In [7, 10, 11], the definition of SI on abstract executions can be separated into the conjunction of UA and CP. Similarly, the definition of PSI on abstract executions can be separated into the conjunction of UA and CC [11]. Interestingly, this is not quite the case for the consistency definitions presented here.

Causal Consistency (CC). This model states that, if a client view includes a version ν written by t prior to committing a transaction, then it must also include the versions which t observes. Clearly, t observes all versions that t reads. Moreover, t observes all previous transactions from the same client. This is captured by $\text{canCommit}_{\text{CC}}$ in Figure 6, defined as $\text{closed}(\mathcal{K}, u, R_{\text{CC}})$ with $R_{\text{CC}} \triangleq \text{SO} \cup \text{WR}_{\mathcal{K}}$. For example, the kv-store of Figure 7e is disallowed by CC: the k_3 version with value v_3 depends on the k_1 version with value v_1 . However, t must have been committed by a client whose view included v_3 of k_3 , but not v_1 of k_1 .

Update Atomic (UA). This consistency model has been proposed in [11] and implemented in [32]. UA disallows concurrent transactions writing to the same key, a property known as *write-conflict freedom*: when two transactions write to the same key, one must see the version written by the other. Write-conflict freedom is enforced by $\text{canCommit}_{\text{UA}}$ which allows

a client to write to key k only if its view includes all versions of k , i.e. its view is closed with respect to the $WW^{-1}(k)$ relation for all keys k written in the fingerprint \mathcal{F} . This prevents the kv-store of Figure 7d, as t and t' concurrently increment the initial version of k by 1. As client views must include the initial versions, once t commits a new version ν with value v_1 to k , then t' must include ν in its view as there is a WW edge from the initial version to ν . As such, when t' increments k , it must read from ν and not the initial version.

Parallel Snapshot Isolation (PSI). This consistency model states that:

1. if a client view includes a version ν written by t prior to committing a transaction, then it must also include the versions that t observes; and
2. there are no write-conflicts.

On abstract executions, where there is a total order over transactions, PSI can be formally defined as the composition of CC and UA [11]. By contrast, it is not possible to define $\text{canCommit}_{\text{PSI}}$ as the conjunction of the $\text{canCommit}_{\text{CC}}$ and $\text{canCommit}_{\text{UA}}$ relations. This is for two reasons. First, the conjunction would only mandate that u be closed with respect to R_{CC} and R_{UA} individually, but not with respect to their union. Recall that closure is defined in terms of the transitive closure of a given relation and thus the closure of R_{CC} and R_{UA} is smaller than the closure of $R_{\text{CC}} \cup R_{\text{UA}}$. As such, we define $\text{canCommit}_{\text{PSI}}$ as closure with respect to R_{PSI} which includes $R_{\text{CC}} \cup R_{\text{UA}}$. Second, recall that CC requires that if a client view includes a version ν written by t' prior to committing a transaction, then it must also include the versions which t' observes. For example, the view of the client of transaction t in Figure 7f must include versions written by t_0 and t_{cl}^1 , satisfying $\text{canCommit}_{\text{CC}}$. Also, recall that UA requires that if a transaction writes to a key k then it must observe all previous versions of k . For example, the client cl' that writes the third version of k_1 in Figure 7f must observe t_{cl}^1 , satisfying $\text{canCommit}_{\text{UA}}$. However, although the client of transaction t observes t_{cl}^1 , it is not able to observe t_{cl}^1 using the combination of CC and UA . This is fixed by including the write-write dependency relation $WW_{\mathcal{K}}$ (e.g. $(t_{cl}^1, t_{cl'}^1) \in WW_{\mathcal{K}}$) in R_{PSI} . Note that Figure 7f shows an example kv-store that satisfies $\text{canCommit}_{\text{CC}}$ and $\text{canCommit}_{\text{UA}}$, but not $\text{canCommit}_{\text{PSI}}$. Under PSI, the view of the client of t should include the versions written by t_{cl}^1 , and therefore read v_3 for key k_2 .

Consistent Prefix (CP). If the total order in which transactions commit is known, then CP can be described as a strengthening of CC [14]: if a client sees the versions written by a transaction t , then it must also see all versions written by transactions that *commit* before t . Although kv-stores only provide *partial* information about the order of transaction commits, this is sufficient to formalise CP.

We can approximate the order in which transactions commit using $WR_{\mathcal{K}}$, $WW_{\mathcal{K}}$, $RW_{\mathcal{K}}$ and SO . This approximation is perhaps best understood in terms of an idealised implementation of CP on a centralised system, where the snapshot of a transaction is determined at its *start point* and its effects are made visible to future transactions at its *commit point*. In this implementation, if $(t, t') \in WR$, then t must commit before t' starts, and hence before t' commits. Similarly, if $(t, t') \in SO$, then t commits before t' starts, and thus before t' commits. Recall that, if $(t'', t') \in RW$, then t'' reads a version that is later overwritten by t' , i.e. t'' cannot see the write of t' , and thus t'' must start before t' commits. As such, if t commits before t'' starts ($(t, t'') \in WR$ or $(t, t'') \in SO$), and $(t'', t') \in RW$, then t must commit before t' commits. In other words, if $(t, t') \in WR; RW$ or $(t, t') \in SO; RW$, then t commits before t' . Finally, if $(t, t') \in WW$, then t must commit before t' . We therefore

define $R_{\text{CP}} \triangleq (\text{WR}_{\mathcal{K}}; \text{RW}_{\mathcal{K}}^? \cup \text{SO}; \text{RW}_{\mathcal{K}}^? \cup \text{WW})$, approximating the order in which transactions commit. As shown in [14], the set $(R_{\text{CP}}^+)^{-1}(t)$ contains all transactions that must be observed by t under CP. We thus define $\text{canCommit}_{\text{CP}}$ by requiring closure with respect to R_{CP} .

The CP model disallows the *long fork anomaly* in Figure 7g, where cl_1 and cl_2 observe the updates to k_1 and k_2 in different orders. Assuming without loss of generality that $t_{cl_1}^2$ commits before $t_{cl_2}^2$, then cl_2 sees the k_1 version with value v_0 before committing $t_{cl_2}^2$. However, as $t \xrightarrow{\text{WR}_{\mathcal{K}}} t_{cl_1}^1 \xrightarrow{\text{SO}} t_{cl_1}^2 \xrightarrow{\text{RW}} t' \xrightarrow{\text{WR}} t_{cl_2}^1$ and $t_{cl_2}^2$ must see the versions written by $t_{cl_2}^1$ before committing, then $t_{cl_2}^2$ must also see the k_1 version with value v_2 , leading to a contradiction.

Snapshot Isolation (SI). On abstract executions, where there is a total order over transactions, SI can be defined as the composition of CP and UA. However, as with PSI, we cannot define $\text{canCommit}_{\text{SI}}$ as the conjunction of their associated canCommit predicates. Rather, we define $\text{canCommit}_{\text{SI}}$ as closure with respect to R_{SI} which includes $R_{\text{CP}} \cup R_{\text{UA}}$. Observe that Figure 7h shows an example kv-store that satisfies $\text{canCommit}_{\text{UA}}$ and $\text{canCommit}_{\text{CP}}$, but not $\text{canCommit}_{\text{SI}}$. Additionally, we include $\text{WW}; \text{RW}$ in R_{SI} . This is because, when the centralised CP implementation (discussed before) is strengthened with write-conflict freedom, then a write-write dependency between transactions t and t' does not only mandate that t commit before t' commits, but also before t' starts. Consequently, if $(t, t') \in \text{WW}; \text{RW}$, then t must commit before t' does.

(Strict) serialisability (SER). Serialisability is the strongest consistency model in settings that abstract from aborted transactions, requiring that transactions execute in a total sequential order. The $\text{canCommit}_{\text{SER}}$ thus allows clients to commit transactions only when their view of the kv-store is complete, i.e. the client view is closed with respect to WW^{-1} . This requirement prevents the kv-store in Figure 7i: if, without loss of generality, t_1 commits before t_2 , then the client committing t_2 must see the k_1 version written by t_1 , and thus cannot read the outdated value v_0 for k_1 .

Weak Snapshot Isolation (WSI): A New Consistency Model. Kv-stores and execution tests are useful for investigating new consistency models. One example is the consistency model induced by combining CP and UA, which we refer to as *Weak Snapshot Isolation (WSI)*. Because WSI is stronger than CP and UA by definition, it forbids all the anomalies forbidden by these consistency models, e.g. the long fork (Figure 7g) and the lost update (Figure 7d). Moreover, WSI is strictly weaker than SI. As such, WSI allows all SI anomalies, e.g. the write skew (Figure 7i), and further allows behaviours not allowed under SI such as that in Figure 7h. The kv-store \mathcal{K} is reachable by executing transactions t_1, t_2, t_3 and t_4 in order. In particular, t_4 is executed using $u = \{k_1 \mapsto \{0\}, k_2 \mapsto \{0, 1\}\}$. However, \mathcal{K} is not reachable under ET_{SI} . This is because t_4 cannot be executed using u under SI: t_4 reads the k_2 version written by t_3 ; but as $(t_2, t_3) \in \text{RW}$ and $(t_1, t_2) \in \text{WW}$, then u should contain the k_1 version written by t_1 , contradicting the fact that t_4 reads the initial version of k_1 . The two consistency models are very similar in that many applications that are correct under SI are also correct under WSI. We give examples of such applications in Section 5.2.

Correctness of ET. Our definitions of consistency models over kv-stores and client views are equivalent to well-known definitions of consistency models over abstract executions [11], and hence over dependency graphs [14]. Given a model M in Figure 6, let $\text{CM}(\text{ET}_M)$ denote the consistency model induced by execution test ET_M of M . For example, when $M = \text{CC}$, then $\text{CM}(\text{ET}_{\text{CC}})$ denotes the consistency model induced by execution test ET_{CC} of CC. Also, let

$\text{CM}(\mathcal{A}_M)$ denote the consistency model of M defined on abstract executions, induced by the set of axioms \mathcal{A}_M [11]. For example, when $M = \text{CC}$, then $\text{CM}(\mathcal{A}_{\text{CC}})$ denotes the consistency mode of CC induced by the CC axioms on abstract executions.

► **Theorem 10.** *For all consistency models M in Figure 6, $\text{CM}(\text{ET}_M) = \text{CM}(\mathcal{A}_M)$.*

The full proof is given in [46], where we define an *intermediate* operational semantics on abstract executions parametrised by axioms, and each step corresponds to an atomic transaction. This is in contrast to [35] which defines a more fine-grained operational semantics.

5 Applications

We use our operational semantics to verify distributed protocols (Section 5.1) and prove invariants of transactional libraries (Section 5.2).

5.1 Application: Verifying Database Protocols

Kv-stores and client views faithfully abstract the state of geo-replicated and partitioned databases, and execution tests provide a powerful abstraction of the synchronisation mechanisms enforced by these databases when committing a transaction. This makes it possible to use our semantics to verify the correctness of distributed database protocols. We demonstrate this by showing that the replicated database, COPS [33], satisfies CC . We refer the reader to [46] for the full details. In [46], we also apply the same method to verify that Clock-SI [21], a partitioned database, satisfies SI .

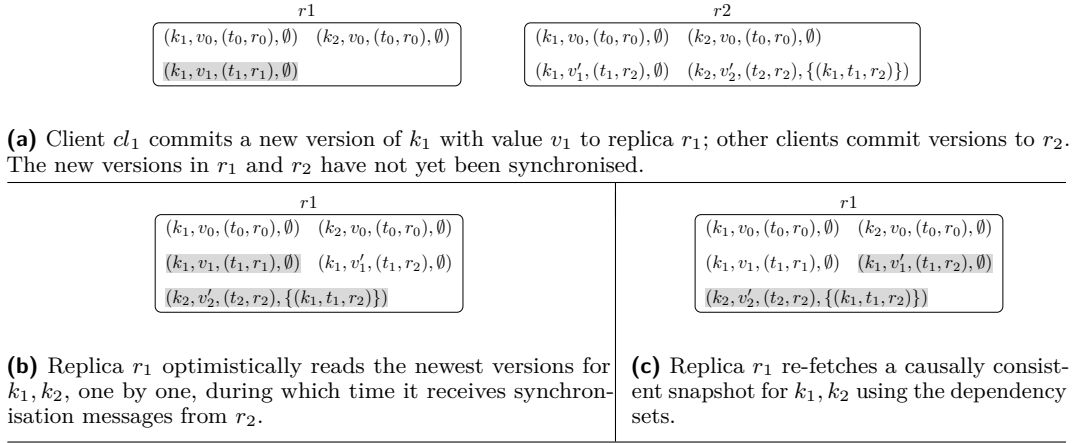
COPS Protocol. COPS is a fully replicated database, with each replica storing multiple versions of each key as shown in Figure 8a. Each COPS version ν such as $(k_1, v_1, (t_1, r_1), \emptyset)$ in Figure 8a, contains a key (k_1), a value (v_1), a *unique* time-stamp (t_1, r_1) denoting when a client first wrote the version to the replica, and a set of dependencies (\emptyset), written $\text{deps}(\nu)$. The time-stamp associated with a version ν has the form (t, r) , where r identifies the replica that committed ν , and t denotes the local time when r committed ν . Each dependency in $\text{deps}(\nu)$ comprises a key and the time-stamp of the versions on which ν directly depends. We define the DEP relation, $(t, r) \xrightarrow{\text{DEP}} (t', r')$, to denote that the version with time-stamp (t, r) is included in the dependency set of the version with time-stamp (t', r') . COPS assumes a total order over replica identifiers. As such, versions can be totally ordered lexicographically.

The COPS API provides two operations:

1. **put** (k, v) for writing to a *single* key k ; and
2. **read** (K) for atomically reading from a *set* of keys K .

Operations from a client are processed by a single replica. Each client maintains a *context*, which is a set of dependencies tracking the versions the client observes.

We demonstrate how a COPS client cl interacts with a replica through the following example: $\text{P}_{\text{COPS}} \triangleq cl : \text{put}(k_1, v_1); \text{read}([k_1, k_2])$. For brevity, we assume that there are two keys, k_1 and k_2 , and two replicas, r_1 and r_2 , where $r_1 < r_2$ (Figure 8a). Initially, client cl connects to replica r_1 and initialises its local context as $ctx = \emptyset$. To execute its first single-write transaction, cl requests to write v_1 to k_1 by sending the message (k_1, v_1, ctx) to its associated replica r_1 and awaits a reply. Upon receiving the message, r_1 produces a monotonically increasing local time t_1 , and uses it to install a new version $\nu = (k_1, v_1, (t_1, r_1), ctx)$, as shown in Figure 8a. Note that the dependency set of ν is the cl context ($ctx = \emptyset$). Replica r_1 then sends the time-stamp (t_1, r_1) back to cl_1 , and cl_1 in turn incorporates (k_1, t_1, r_1) in its local



■ **Figure 8** COPS protocol.

context, i.e. cl observes its own write. Finally, r_1 propagates the written version to other replicas *asynchronously* by sending a *synchronisation message* using *causal delivery*: when a replica r' receives a version ν' from another replica r , it waits for all ν' dependencies to arrive at r' , and then accepts ν' . As such, the set of versions contained in each replica is closed with respect to the DEP relation. In the example above, when other replicas receive ν from r_1 , they can immediately accept ν as $\text{deps}(\nu) = \emptyset$. Note that replicas may accept new versions from different clients in parallel.

To execute its second multi-read transaction, client cl requests to read from the k_1, k_2 keys by sending the message $\{k_1, k_2\}$ to replica r_1 and awaits a reply. Upon receiving this message, r_1 builds a *DEP-closed snapshot* (a mapping from $\{k_1, k_2\}$ to values) in two phases as follows. First, r_1 *optimistically reads* the most recent versions for k_1 and k_2 , *one at a time*. This process may be interleaved with other writes and synchronisation messages. For instance, Figure 8b depicts a scenario where r_1 :

1. first reads $(k_1, v_1, (t_1, r_1), \emptyset)$ for k_1 (highlighted);
2. then receives two synchronisation messages from r_2 , containing versions $(k_1, v'_1, (t_1, r_2), \emptyset)$ and $(k_2, v'_2, (t_2, r_2), \{(k_1, t_1, r_2)\})$; and
3. finally reads $(k_2, v'_2, (t_2, r_2), \{(k_1, t_1, r_2)\})$ for k_2 (highlighted).

As such, the current snapshot for $\{k_1, k_2\}$ are not DEP-closed: $(k_2, v'_2, (t_2, r_2), \{(k_1, t_1, r_2)\})$ depends on a k_1 version with time-stamp (t_1, r_2) which is bigger than (t_1, r_1) for k_1 . To remedy this, after the first phase of optimistic reads, r_1 combines (unions) all dependency sets of the versions from the first phase as a *re-fetch set*, and uses it to *re-fetch* the most recent version of each key with the biggest time-stamp from the union of the re-fetch set and the versions from the first phase. For instance, in Figure 8c, replica r_1 re-fetches the newer version $(k_1, v'_1, (t_1, r_2), \emptyset)$ for k_1 . Finally, the snapshot obtained after the second phase is sent to the client, where it is added to the client context. For their specific setting, Lloyd et al. [33] informally argue that the snapshot sent to the client is causally consistent. By contrast, in what follows we verify the COPS protocol with our general definition of CC.

COPS Verification. We define an operational semantics for the COPS protocol, which uses fine-grained single reads and writes of a key. Using our semantics, we then show that COPS traces can be refined to traces in our semantics using ET_{CC} in three steps:

$$\Theta_0 \xrightarrow{cl, r_1: (W, k_1, (t_1, r_1))} \Theta_1 \xrightarrow{cl, r_1: \mathbf{s}} \Theta_2 \xrightarrow{cl, r_1: (R, k_1, (t_1, r_1))} \Theta_3 \xrightarrow{r_1: \mathbf{sync}} \Theta_4 \xrightarrow{cl, r_1: (R, k_2, (t_2, r_2))} \Theta_5 \xrightarrow{cl, r_1: \mathbf{p}} \Theta_6 \xrightarrow{\iota} \Theta_7 \xrightarrow{cl, r_1: (R, k_1, (t_1, r_2))} \Theta_8 \xrightarrow{\iota'} \Theta_9 \xrightarrow{cl, r_1: (R, k_2, (t_2, r_2))} \Theta_{10} \xrightarrow{cl, r_1: \mathbf{e}} \dots$$

(a) The COPS trace that produces Figures 8b and 8c.

$$\Theta'_5 \xrightarrow{\iota} \Theta'_6 \xrightarrow{\iota'} \Theta'_7 \xrightarrow{cl, r_1: \mathbf{p}} \Theta'_8 \xrightarrow{cl, r_1: (R, k_1, (t_1, r_2))} \Theta'_9 \xrightarrow{cl, r_1: (R, k_2, (t_2, r_2))} \Theta'_{10} \xrightarrow{cl, r_1: \mathbf{e}} \dots$$

(b) The normalised COPS trace.

$$k_1 \mapsto \begin{array}{|c|c|c|} \hline (t_0, r_0) & (t_1, r_1) & (t_1, r_2) \\ \hline v_0 & v_1 & v'_1 \\ \hline - & - & - \\ \hline \end{array} \xrightarrow{k_2 \mapsto} \begin{array}{|c|c|} \hline (t_0, r_0) & (t_2, r_2) \\ \hline v_0 & v'_2 \\ \hline - & - \\ \hline \end{array} \xrightarrow{cl, u : \{k_1 \mapsto \{0, 1, 2\}, k_2 \mapsto \{0, 1\}\}, \mathcal{F} : \{(R, k_1, v'_1), (R, k_2, v'_2)\}} \begin{array}{|c|c|c|} \hline (t_0, r_0) & (t_1, r_1) & (t_1, r_2) \\ \hline v_0 & v_1 & v'_1 \\ \hline - & - & - \cup \{t_{rd}\} \\ \hline \end{array} \xrightarrow{k_2 \mapsto} \begin{array}{|c|c|} \hline (t_0, r_0) & (t_2, r_2) \\ \hline v_0 & v'_2 \\ \hline - & - \cup \{t_{rd}\} \\ \hline \end{array}$$

(c) The step encoding the multi-read transaction depicted above: the kv-store before update encodes Figure 8a, and the views (highlighted) encoding of the client contexts before and after the update.

■ **Figure 9** COPS traces and trace refinement.

1. every COPS trace can be transferred to an equivalent normalised COPS trace, in which multiple reads of a transaction are not interleaved by other transactions; and
2. the normalised COPS trace can be refined to a trace in our semantics, in which
3. each step satisfies ET_{CC} .

The COPS operational semantics describes transitions over abstract states Θ comprising a set of replicas, a set of client contexts and a program. For instance, the COPS trace that produces Figures 8b and 8c is depicted in Figure 9a, stating that given client cl and replica r_1 ,

1. cl writes version $(W, k_1, (t_1, r_1))$ to r_1 ;
2. cl starts a multi-read transaction (\mathbf{s});
3. cl reads $(R, k_1, (t_1, r_1))$ from r_1 ;
4. r_1 receives synchronisation messages (\mathbf{sync});
5. cl reads $(R, k_2, (t_2, r_2))$ from r_1 ;
6. cl enters the second re-fetch phase of the multi-read transaction (\mathbf{p});
7. an arbitrary step ι interferes;
8. cl re-fetches version $(R, k_1, (t_1, r_2))$ from r_2 and puts it in the snapshot;
9. an arbitrary step ι' interferes;
10. cl puts the version $(R, k_2, (t_2, r_2))$ in the snapshot; and
11. cl reads the values in the snapshot and commits the transaction (\mathbf{e}).

Recall that a multi-read transaction does not execute atomically in the replica, as captured by multiple read transitions in the trace. For example, steps ι and ι' in Figure 9a interleave the multi-read transaction of cl . Note that the optimistic reads are not observable by the client and thus it suffices to show that the reads from the second re-fetch phase are atomic. To show this, we *normalise* the trace as follows. For each multi-read transaction, we move the reads in the re-fetch phase to the right towards the return step \mathbf{e} , so that they are no longer interleaved by others. An example of a normalised trace is given in Figure 9b. In each

multi-read transaction, the re-fetch phase can only read a version committed before the p step. For example, in Figure 9a (top) the multi-read transaction of cl can only read versions in Θ_5 and before. As such, normalising does not alter the returned versions of transactions. After normalisation, transactions in the resulting trace appear to execute atomically.

We next show that a normalised COPS trace can be refined to a trace in our operational semantics. To do this, we encode an abstract COPS state Θ as a configuration in our semantics (Figure 9c). We map all the COPS replicas to a single kv-store. The writer of a version in the kv-store is uniquely determined by the time-stamp of the corresponding COPS version, while the reader set is given by creating new transaction identifiers for the read-only transactions such as the identifier t_{rd} in Figure 9c. For example, the COPS state in Figure 8a can be encoded as the kv-store depicted in Figure 9c. Since the context of a client cl identifies the set of COPS versions that cl sees, we can project COPS client contexts to our client views over kv-stores. For example, the contexts of cl before and after committing its second multi-read transaction in P_{COPS} is encoded as the client views depicted in Figure 9c.

We finally show that every step in the kv-store trace satisfies ET_{CC} . Note that existing verification techniques [11, 16] require examining the *entire* sequence of operations of a protocol to show that it implements a consistency model. By contrast, we only need to look at how the state evolves after a *single* transaction is executed. In particular, we check the client views over the kv-store. Intuitively, we observe that when a COPS client cl executes a transaction then:

1. the cl context grows, and thus we obtain a more up-to-date view of the associated kv-store, i.e. $vShift_{MR}$ holds;
2. the cl context always includes the time-stamp of the versions written by itself, and thus the corresponding client view always includes the versions cl has written, i.e. $vShift_{RW}$ holds and
3. the cl context is always closed to the relation DEP , which contains the relation $SO \cup WR_{\mathcal{K}}$, i.e. $closed(\mathcal{K}, u, R_{CC})$ holds.

We have thus demonstrated that COPS satisfies CC (see [46] for the full details).

5.2 Application: Invariant Properties of Transactional Libraries

With our operational semantics, we are able to prove invariant properties of kv-stores, such as: the robustness of the single counter library against PSI ; the robustness of a multi-counter library (Section 2) and the well-known banking library [2] against SI ; and the correctness of a lock library against UA and hence PSI , even though the lock library is not robust for these consistency models. The robustness of the multi-counter and banking library follow from a general proof of the robustness of the so-called WSI -safe libraries against WSI , and hence SI . Our robustness results are the first to be proved for client sessions, in contrast with static analysis techniques for checking robustness [7, 12, 14, 35] that did not support client sessions.

Single-counter Library: Robustness. A *transactional library* is a set of transactional operations, e.g. the counter library, $\text{Counter}(\mathbf{k}) \triangleq \{\text{Inc}(\mathbf{k}), \text{Read}(\mathbf{k})\}$, given in Section 2. Client programs of the transactional library can access the underlying kv-store using only the operations of the library. A transactional library is *robust* against an execution test ET if, for all client programs P of the library, the kv-stores \mathcal{K} obtained under ET can also be obtained under SER , i.e. given initial kv-store \mathcal{K}_0 , initial view environment \mathcal{U}_0 and an arbitrary client environment \mathcal{E} , for any reachable kv-store \mathcal{K} such that $(\mathcal{K}_0, \mathcal{U}_0, \mathcal{E}), P \xrightarrow{*}_{ET} (\mathcal{K}, _, _)$, $_, _ _$, then $\mathcal{K} \in \text{CM}(SER)$. Our robustness results use the following theorem (Theorem 11) that a kv-stores obtained under a trace satisfies serialisability if and only if it contains no cycles.

► **Theorem 11.** A kv-store $\mathcal{K} \in \text{CM}(\text{SER})$ iff $(\text{SO} \cup \text{WR}_{\mathcal{K}} \cup \text{WW}_{\mathcal{K}} \cup \text{RW}_{\mathcal{K}})^+ \cap \text{Id} = \emptyset$.

► **Theorem 12.** The single counter library, $\text{Counter}(\mathbf{k}) \triangleq \{\text{Inc}(\mathbf{k}), \text{Read}(\mathbf{k})\}$ given in Section 2, is robust against PSI.

Proof (sketch). In the single-counter library, $\text{Counter}(\mathbf{k})$, a client reads from k by calling $\text{Read}(\mathbf{k})$, and writes to k by calling $\text{Inc}(\mathbf{k})$ which first reads the value of k and subsequently increments it by one. As PSI enforces write-conflict freedom (**UA**), we know that if a transaction t updates k (via $\text{Inc}(\mathbf{k})$) and writes version ν to k , then it must have read the version of k immediately preceding ν : $\forall t, i > 0. t = \text{w}(\mathcal{K}(k, i)) \Rightarrow t \in \text{rs}(\mathcal{K}(k, i-1))$. Moreover, as PSI enforces monotonic reads (**MR**), the order in which clients observe the versions of k (via $\text{Read}(\mathbf{k})$) is consistent with the order of versions in $\mathcal{K}(k)$. As such, the invariant illustrated below always holds (i.e. the kv-store is always has the depicted shape), where $\{t_i\}_{i=1}^n$ and $\bigcup_{i=0}^n T_i$ denote disjoint sets of transactions calling $\text{Inc}(\mathbf{k})$ and $\text{Read}(\mathbf{k})$, respectively:

$$(0, t_0, T_0 \cup \{t_1\}) :: (1, t_1, T_1 \cup \{t_2\}) :: \dots \quad \left| \quad k \mapsto \begin{array}{|c|c|c|c|c|} \hline 0 & t_0 & t_1 & \dots & t_{n-1} & t_n \\ \hline T_0 \uplus \{t_1\} & T_1 \uplus \{t_2\} & \dots & T_{n-1} \uplus \{t_n\} & T_n \\ \hline \end{array} \right.$$

We define the \dashrightarrow relation depicted above by extending the relation $R \triangleq \text{SO} \cup \{(t, t') \mid \exists i. (t = t_i \wedge (t' = t_{i+1} \vee t' \in T_i)) \vee (t \in T_i \wedge t' = t_{i+1})\}$ to a strict total order (i.e. a total, irreflexive and transitive relation). Note that \dashrightarrow contains $\text{SO} \cup \text{WR}_{\mathcal{K}} \cup \text{WW}_{\mathcal{K}} \cup \text{RW}_{\mathcal{K}}$ and thus $(\text{SO} \cup \text{WR}_{\mathcal{K}} \cup \text{WW}_{\mathcal{K}} \cup \text{RW}_{\mathcal{K}})^+$ is irreflexive, i.e. $\text{Counter}(\mathbf{k})$ is robust against PSI. By contrast, a multi-counter library on a set of keys K , $\text{Counters}(K) \triangleq \bigcup_{k \in K} \text{Counter}(\mathbf{k})$, is *not* robust against PSI. Recall from Section 2 that unlike in **SER** and **SI**, clients of the multi-counter library under PSI can observe the increments on different keys in different orders (see Figure 7g). Hence, the multi-counter library is not robust against PSI. ◀

WSI-safe Libraries: Robustness. Our next task is to show that the multi-counter library and the banking library from [2] are robust against **SI**. We do this by defining the notion of **WSI-safe** transactional libraries, and proving a general robustness result for such libraries against **WSI**, and thus **SI**. The proof of this general result uses the following two acyclic properties of kv-stores, where ET_{\top} is the most permissive execution test (Definition 9).

► **Theorem 13.** Any kv-store $\mathcal{K} \in \text{CM}(\text{ET}_{\top})$ satisfies $(\text{SO} \cup \text{WR}_{\mathcal{K}})^+ \cap \text{Id} = \emptyset$.

Proof (sketch). From the definition of **CM** (Definition 9) we know a kv-store $\mathcal{K} \in \text{CM}(\text{ET}_{\top})$ must be reachable with a given program. This means that Theorem 13 can be seen as an invariant property. We prove it by induction on the length of a trace. For the base case, the initial kv-store \mathcal{K}_0 trivially contains no cycles. For the inductive case, since local computation steps do not rely on the kv-store, let us focus on the case where the last transaction step has the form: $(\mathcal{K}, \mathcal{U}, \mathcal{E}), \mathcal{P} \xrightarrow{(cl, u, \mathcal{F})_{\text{ET}}} (\mathcal{K}', \mathcal{U}', \mathcal{E}'), \mathcal{P}'$, where \mathcal{K} contains no $R \triangleq (\text{SO} \cup \text{WR}_{\mathcal{K}})$ cycles by the inductive hypothesis. Let t be the new transaction in \mathcal{K}' . We then proceed by contradiction and assume that \mathcal{K}' has a R cycle. As \mathcal{K} contains no R cycles, this cycle must involve t , i.e. $t \xrightarrow{R} t_1 \xrightarrow{R} \dots \xrightarrow{R} t_n \xrightarrow{R} t$, where t_1, \dots, t_n are distinct. As t is the last transaction and $t \notin \mathcal{K}$, we cannot have $t \xrightarrow{\text{SO}} t_1$. Similarly, all versions written by t have empty reader sets, and thus we cannot have $t \xrightarrow{\text{WR}_{\mathcal{K}'}} t_1$. This then leads to a contradiction as $t \xrightarrow{\text{SO} \cup \text{WR}_{\mathcal{K}'}} t_1$. Therefore, the new kv-store \mathcal{K}' satisfies $(\text{SO} \cup \text{WR}_{\mathcal{K}'})^+ \cap \text{Id} = \emptyset$. ◀

► **Theorem 14.** Any kv-store $\mathcal{K} \in \text{CM}(\text{ET}_{\text{CP}})$ satisfies $((\text{SO} \cup \text{WR}_{\mathcal{K}}); \text{RW}_{\mathcal{K}}^?)^+ \cap \text{Id} = \emptyset$.

Proof (sketch). We proceed as in the proof of Theorem 13. For the inductive case, consider $(\mathcal{K}, \mathcal{U}, \mathcal{E}), \mathcal{P} \xrightarrow{(cl, u, \mathcal{F})}_{\text{ET}} (\mathcal{K}', \mathcal{U}', \mathcal{E}'), \mathcal{P}'$, where \mathcal{K} contains no $R \triangleq ((\text{SO} \cup \text{WR}_{\mathcal{K}}); \text{RW}_{\mathcal{K}}^2)$ cycles by the inductive hypothesis. Let us then assume \mathcal{K}' has a R cycle which must include the new transaction t . There are then two cases as follows where t_1, \dots, t_n are distinct:

$$1. t \xrightarrow{R} t_1 \xrightarrow{R} \dots \xrightarrow{R} t_n \xrightarrow{R} t$$

This cycle cannot exist as t is the last transaction in \mathcal{K}' . More concretely, as in Theorem 13 we know we cannot have $t \xrightarrow{\text{SO}} t_1$ or $t \xrightarrow{\text{WR}_{\mathcal{K}'}} t_1$. For analogous reasons, we cannot have $t \xrightarrow{\text{SO}} t' \xrightarrow{\text{RW}_{\mathcal{K}'}} t_1$ or $t \xrightarrow{\text{WR}_{\mathcal{K}'}} t' \xrightarrow{\text{RW}_{\mathcal{K}'}} t_1$, for some transaction $t' \in \mathcal{K}$.

$$2. t_1 \xrightarrow{R} \dots \xrightarrow{R} t_n \xrightarrow{(\text{SO} \cup \text{WR}_{\mathcal{K}'})} t \xrightarrow{\text{RW}_{\mathcal{K}'}} t_1$$

From ET_{CP} the view u of t must contain all versions written by t_1, \dots, t_n . As such, we cannot have $t \xrightarrow{\text{RW}_{\mathcal{K}'}} t_1$ as by $\text{RW}_{\mathcal{K}'}$ we know u is behind the versions written by t_1 . ◀

Specific libraries [2, 5, 7] have been shown to be robust against SI by individually checking all final results of all their client programs. By contrast, we identify the notion of a *WSI-safe* library and prove that such a library is robust against WSI, and hence SI, by showing that the acyclic invariant given in Theorem 11 is preserved by each transition step.

► **Definition 15 (WSI-safe).** A library is *WSI-safe* if and only if, for all its client programs \mathcal{P} and all kv-stores \mathcal{K} , if \mathcal{K} is obtained by executing \mathcal{P} under WSI^7 , then for all t, k, i, i' :

$$t \in \text{rs}(\mathcal{K}(k, i)) \wedge t \neq \text{w}(\mathcal{K}(k, i')) \Rightarrow \forall k', j. t \neq \text{w}(\mathcal{K}(k', j)), \quad (1)$$

$$t \neq t_0 \wedge t = \text{w}(\mathcal{K}(k, i)) \Rightarrow \exists j. t \in \text{rs}(\mathcal{K}(k, j)), \quad (2)$$

$$t \neq t_0 \wedge t = \text{w}(\mathcal{K}(k, i)) \wedge \exists k', j, j'. t \in \text{rs}(\mathcal{K}(k', j)) \Rightarrow t = \text{w}(\mathcal{K}(k', j')). \quad (3)$$

That is, (1) if a transaction t reads from k but does not write to it, then t must be a read-only transaction; (2) if t writes to k , then it must also read from it, a property known as *no-blind writes*⁸; and (3) if t writes to k , then it must also write to all keys it reads from. The read-only transactions, satisfying (1), can be reordered to be next to the write that they are reading. Their behaviour is, thus, serialisable in that the write they are reading is current. Under WSI and SI, transactions satisfying *strict no-blind writes* (i.e. (2) and (3)) enforce a total order over transactions on a key, which is enough to obtain serialisable behaviour.

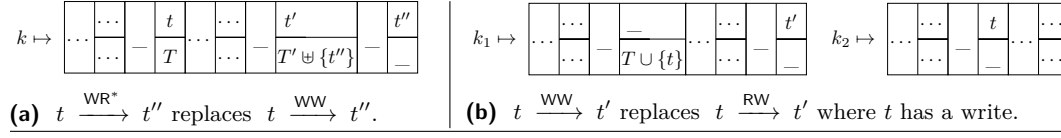
It is straightforward to see that the multi-counter library given in Section 2 is *WSI-safe*; we will show that the banking example in [2] is *WSI-safe*. The example in [7] is *WSI-safe*. In [5], there are many examples of libraries that are shown to be robust against SI: the smaller examples are *WSI-safe*; the larger examples have not been checked.

► **Theorem 16 (WSI robustness).** A *WSI-safe* library is robust against WSI.

Proof (sketch). Pick a *WSI-safe* library L , a client program \mathcal{P} of L and a kv-store \mathcal{K} obtained from executing \mathcal{P} under WSI, i.e. $(\mathcal{K}_0, \mathcal{U}_0, \mathcal{E}), \mathcal{P} \xrightarrow{*}_{\text{ET}_{\text{WSI}}} (\mathcal{K}, _, _), _$. From Theorem 11 it suffices to prove that $(\text{SO} \cup \text{WR}_{\mathcal{K}} \cup \text{WW}_{\mathcal{K}} \cup \text{RW}_{\mathcal{K}})^+$ is acyclic. We proceed by contradiction. Let us assume there exists t_1 such that $t_1 \xrightarrow{(\text{SO} \cup \text{WR}_{\mathcal{K}} \cup \text{WW}_{\mathcal{K}} \cup \text{RW}_{\mathcal{K}})^+} t_1$. From Theorem 13 we know $(\text{SO} \cup \text{WR}_{\mathcal{K}})^+$ is acyclic. Moreover, thanks to *no-blind-writes* in (2) and **UA**, any $\text{WW}_{\mathcal{K}}(k)$ edge on a key k can be replaced by $\text{WR}_{\mathcal{K}}^+(k)$, as illustrated in Figure 10a. As

⁷ That is, for initial kv-store \mathcal{K}_0 , initial view environment \mathcal{U}_0 and arbitrary client environment \mathcal{E} , $(\mathcal{K}_0, \mathcal{U}_0, \mathcal{E}), \mathcal{P} \xrightarrow{*}_{\text{ET}_{\text{WSI}}} (\mathcal{K}, _, _), _$.

⁸ From **UA**, it is immediate that $j = i - 1$.



■ **Figure 10** WSI-safety.

such, $(SO \cup WR_{\mathcal{K}})^+ \cup WW_{\mathcal{K}}$ is acyclic and thus this cycle is of the form: $t_1 \xrightarrow{R^*} \xrightarrow{RW} \xrightarrow{R^*} \dots \xrightarrow{R^*} \xrightarrow{RW} \xrightarrow{R^*} t_1$, where $R \triangleq SO \cup WR \cup WW$. From (3) we know an $RW_{\mathcal{K}}(k_1)$ edge on a key k_1 starting from a writing transaction t can be replaced by a WW edge, as illustrated in Figure 10b. Moreover, from (2) we know we can replace WW edges by WR^+ . We thus have: $t_1 \xrightarrow{R'^*} \xrightarrow{RW} \xrightarrow{R'^+} \dots \xrightarrow{R'^+} \xrightarrow{RW} \xrightarrow{R'^*} t_1$, where $R' \triangleq SO \cup WR$, i.e. $t_1 \xrightarrow{(R'; RW^?)*} t_1$. This, however, leads to a contradiction by Theorem 14. ◀

Using Theorem 16, we can prove the robustness of the banking library in [2] against WSI, and hence SI. Alomari et al. [2] informally showed that this example is robust: they identified a notion of dangerous dependency between transactions which, they argued, can lead to violation of robustness of SI; and they argued that this banking example contains no such dangerous dependencies. The original banking example worked with a relational database with three tables *account*, *saving* and *checking*. The *account* table maps customer names to customer IDs ($\text{Account}(\text{Name}, \text{CID})$); the *saving* table maps customer IDs to their saving balances ($\text{Saving}(\text{CID}, \text{Balance})$); and the *checking* table maps customer IDs to their checking balances ($\text{Checking}(\text{CID}, \text{Balance})$). The balance of a saving account must be non-negative, but a checking account may have a negative balance.

For simplicity, we encode the saving and checking tables as a single kv-store, and omit the *account* table as it is an immutable lookup table. We model a customer ID as an integer $n \in \mathbb{N}$, and assume that the balances are integer values. We then define the key associated with customer n in the checking table as $n_c \triangleq 2n$, and define the key associated with n in the saving table as $n_s \triangleq 2n+1$, i.e. $\text{KEY} \triangleq \bigcup_{n \in \mathbb{N}} \{n_c, n_s\}$. Moreover, if n identifies a customer with $(_, n) \in \text{Account}(\text{Name}, \text{CID})$, then $(n, \text{val}(\mathcal{K}(n_s, |\mathcal{K}(n_s)| - 1))) \in \text{Saving}(\text{CID}, \text{Balance})$ and $(n, \text{val}(\mathcal{K}(n_c, |\mathcal{K}(n_c)| - 1))) \in \text{Checking}(\text{CID}, \text{Balance})$.

The banking library provides five transactional operations:

$$\begin{aligned} \text{balance}(\mathbf{n}) &\triangleq [x := [n_c]; y := [n_s]; \text{ret} := x + y] \\ \text{depositCheck}(\mathbf{n}, v) &\triangleq [\text{if } (v \geq 0) \{ x := [n_c]; [n_c] := x + v \}] \\ \text{transactSaving}(\mathbf{n}, v) &\triangleq [x := [n_s]; \text{if } (v + x \geq 0) \{ [n_s] := x + v \}] \\ \text{amalgamate}(\mathbf{n}, \mathbf{n}') &\triangleq \begin{bmatrix} x := [n_s]; y := [n_c]; z := [n'_c]; \\ [n_s] := 0; [n_c] := 0; [n'_c] := x + y + z \end{bmatrix} \\ \text{writeCheck}(\mathbf{n}, v) &\triangleq \begin{bmatrix} x := [n_s]; y := [n_c]; \\ \text{if } (v > 0 \ \&\& \ x + y < v) \{ [n_c] := y - v - 1 \} \\ \text{else} \{ [n_c] := y - v \} \quad [n_s] := x \end{bmatrix} \end{aligned}$$

The $\text{balance}(\mathbf{n})$ operation returns the total balance of customer \mathbf{n} in ret . The $\text{depositCheck}(\mathbf{n}, v)$ deposits v to the checking account of customer \mathbf{n} when v is non-negative, otherwise it leaves the checking account unchanged. When $v \geq 0$, $\text{transactSaving}(\mathbf{n}, v)$ deposits v to the saving account of \mathbf{n} . When $v < 0$, $\text{transactSaving}(\mathbf{n}, v)$ withdraws v from the saving account of \mathbf{n} only if the resulting balance is non-negative, otherwise the saving account remains unchanged. The $\text{amalgamate}(\mathbf{n}, \mathbf{n}')$ operation moves the combined checking and

saving balance of customer n to the checking account of customer n' . Lastly, `writeCheck(n, v)` cashes a cheque of customer n in the amount v by deducting v from its checking account. If n does not hold sufficient funds (i.e. the combined checking and saving balance is less than v), customer n is penalised by deducting one additional pound. In [2], the authors argue that to make this library robust against SI, the `writeCheck(n, v)` operation must be strengthened by writing back the saving account balance (via $[n_s] := x$), even though this is unchanged.

The banking library is more complex than the multi-counter library. Nevertheless, all banking transactions are either read-only or satisfy the no-blind writes property. Hence, the banking library is WSI-safe, and so robust against WSI and SI.

Lock Library: Mutual-exclusion Guarantee. Finally, we demonstrate that, although a distributed lock library is not robust against UA, we can nevertheless prove an invariant property stating that only one client can hold the lock at a given time, thus establishing a mutual exclusion guarantee. The distributed lock library provides the following operations on a key k :

$$\begin{aligned} \text{tryLock}(k) &\triangleq [x := [k]; \text{ if } (x=0)\{ [k] := \text{ClientID}; m := \text{true} \}\text{ else}\{ m := \text{false} \}] \\ \text{lock}(k) &\triangleq \text{ do}\{ \text{tryLock}(k) \}\text{ until}(m=\text{false}) \quad \text{unlock}(k) \triangleq [[k] := 0] \end{aligned}$$

The `tryLock` operation reads the k value; if the value is zero (i.e. the lock is available), then it sets it to the client ID and returns `true`; otherwise it leaves it unchanged and returns `false`. The `lock` operation calls `tryLock` until it successfully acquires the lock. The `unlock` operation simply set the k value to zero.

Consider the program P_{LK} where clients cl and cl' compete to acquire the lock k :

$$P_{LK} \triangleq (cl : (\text{lock}(k); \dots; \text{unlock}(k))^* \parallel cl' : (\text{lock}(k); \dots; \text{unlock}(k))^*)$$

The locking program in P_{LK} is correct, in that only one client can hold the lock at a time, when executed under serialisability. Since all the operations are trivially WSI-safe, P_{LK} is robust and hence correct under WSI as well as stronger models such as SI. However, P_{LK} is not robust under UA or PSI: `lock` may read an old value of key k until it reads its most up-to-date value and acquires it. Nevertheless, we show that P_{LK} is correct under UA (and hence PSI) in that it satisfies a mutual exclusion guarantee where only one client can hold the lock at a time. We capture this guarantee by the following invariant, stating that for all positive i ($i > 0$):

$$\text{val}(\mathcal{K}(k, i)) \neq 0 \Leftrightarrow \text{val}(\mathcal{K}(k, i - 1)) = 0 \tag{4}$$

$$\text{val}(\mathcal{K}(k, i)) = 0 \Rightarrow \text{w}(\mathcal{K}(k, i)) = \text{w}(\mathcal{K}(k, i - 1)) \tag{5}$$

It is straightforward to show that, under UA, only one client can hold the lock (4), and the same client releases the lock (5). Assume a kv-store \mathcal{K} satisfies this invariant. Given the lock program in P_{LK} , if the latest value of k is 0, then all clients are competing to acquire k , and thanks to UA only a client cl with full view of k can install a new version with its unique client ID. This will stop other clients from acquiring k as the latest value is now non-zero. Subsequently, when cl executes its next transaction, i.e. `unlock(k)`, it releases the lock and installs a new version with value zero.

Invariants vs. Execution Graphs. We have demonstrated how invariant properties of transactional libraries can be used to prove their robustness, as well as library-specific guarantees such as mutual exclusion. Although existing work can establish the robustness of

a library using execution graphs (e.g. dependency graphs of [1]), they typically do this by checking the *final* results of all its client programs. By contrast, thanks to our operational model, we achieve this by establishing an invariant property at each execution step, thus allowing a simpler, more compositional proof. Moreover, whilst it is straightforward for us to prove library-specific guarantees (e.g. mutual exclusion for locks) by simply encoding them as an invariant of the library, establishing such properties using execution graphs is much more difficult. This is because execution graphs do not directly record the library *state* and merely record the execution shape, thus making it harder to reason about such guarantees.

6 Conclusions and Future Work

We have introduced an interleaving operational semantics for describing the client-observable behaviour of atomic transactions over distributed kv-stores, using abstract states comprising global, centralised kv-stores, partial client views, and transition steps parametrised by an execution test which directly captures when a transaction is able to commit on a state. Using these execution tests, we provide a general definition of consistency model and provide example instantiations including **CC**, **PSI**, **SI** and **SER**. In [46], we prove that our definitions are equivalent to the existing definitions in the literature that use execution graphs [11].

We have used our semantics to verify that protocols of real-world distributed databases satisfy particular consistency models, e.g. that the replicated database COPS [33] satisfies **CC**, and the partitioned database Clock-SI [21] satisfies **SI**. These results contrast with those of [21, 33], which justify the correctness of implementations using consistency model definitions that are specific to the implementations. We have also proved several invariant properties for clients, showing that the clients of several libraries (single-counter, multi-counter and banking libraries) are robust against the appropriate models, and showing that certain clients of a lock library satisfy a mutual exclusion property under **PSI**, even though they are not robust against **PSI**. We thus believe that our semantics provides an interesting abstract interface between distributed implementations and clients. We plan to validate further the usefulness of our semantics by verifying other well-known protocols of distributed databases [4, 30, 34, 43], exploring robustness results for OLTP workloads such as TPC-C [44] and RUBiS [39], and exploring other program analysis techniques such as transaction chopping [13, 41], invariant checking [24, 47] and program logics [27]. We also plan to develop tools to generate litmus tests for implementations and to analyse client programs.

Our work assumes the *snapshot property* and the *last-write-wins* policy, common assumptions in real-world distributed databases. Under these assumptions, we are not aware of a consistency model that we cannot express using our semantics. There are consistency models that do not satisfy these assumptions, e.g. *read committed* [4] captured in [16]. In future, we will explore whether it is possible to weaken our assumptions to express such weak consistency models. This might be possible by introducing “promises” in the style of [28].

There are many resonances between the high-level behaviour of distributed systems and the low-level behaviour of weak memory. Indeed, our partial client views were inspired by the views of the “promising” C11 semantics in [28]. In future, we plan to explore whether our semantics of atomic transactions can be loosened to describe the more fine-grained behaviour of transactions on weak memory [38, 15]. We are also interested in the work of Doherty et al. [20], describing an operational semantics and a program logic for the release-acquire (RA) fragment of C11, which, interestingly, is based on dependency graphs. We believe that we can adapt our semantics to model the RA fragment, using simple read-write primitives rather than atomic transactions and a variant of our definition of causal consistency.

References

- 1 Atul Adya. *Weak Consistency: A Generalized Theory and Optimistic Implementations for Distributed Transactions*. PhD thesis, Massachusetts Institute of Technology, Cambridge, MA, USA, 1999. URL: <http://pmg.csail.mit.edu/papers/adya-phd.pdf>.
- 2 M. Alomari, M. Cahill, A. Fekete, and U. Rohm. The cost of serializability on platforms that use snapshot isolation. In *2008 IEEE 24th International Conference on Data Engineering*, pages 576–585, April 2008. doi:10.1109/ICDE.2008.4497466.
- 3 Masoud Saeida Ardekani, Pierre Sutra, and Marc Shapiro. Non-monotonic snapshot isolation: Scalable and strong consistency for geo-replicated transactional systems. In *Proceedings of the 2013 IEEE 32nd International Symposium on Reliable Distributed Systems, SRDS '13*, page 163–172, USA, 2013. IEEE Computer Society. doi:10.1109/SRDS.2013.25.
- 4 Peter Bailis, Alan Fekete, Ali Ghodsi, Joseph M. Hellerstein, and Ion Stoica. Scalable atomic visibility with ramp transactions. *ACM Trans. Database Syst.*, 41(3), July 2016. doi:10.1145/2909870.
- 5 Sidi Mohamed Beillahi, Ahmed Bouajjani, and Constantin Enea. Checking robustness against snapshot isolation. *CoRR*, abs/1905.08406, 2019. arXiv:1905.08406.
- 6 Hal Berenson, Phil Bernstein, Jim Gray, Jim Melton, Elizabeth O’Neil, and Patrick O’Neil. A Critique of ANSI SQL Isolation Levels. In *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data, SIGMOD’95*, pages 1–10. ACM, 1995. doi:10.1145/223784.223785.
- 7 Giovanni Bernardi and Alexey Gotsman. Robustness against Consistency Models with Atomic Visibility. In *Proceedings of the 27th International Conference on Concurrency Theory*, pages 7:1–7:15, 2016. doi:10.4230/LIPIcs.CONCUR.2016.7.
- 8 Carsten Binnig, Stefan Hildenbrand, Franz Färber, Donald Kossmann, Juchang Lee, and Norman May. Distributed Snapshot Isolation: Global Transactions Pay Globally, Local Transactions Pay Locally. *The VLDB Journal*, 23(6):987–1011, December 2014. doi:10.1007/s00778-014-0359-9.
- 9 Sebastian Burckhardt, Manuel Fahndrich, Daan Leijen, and Mooly Sagiv. Eventually Consistent Transactions. In *Proceedings of the 21st European Symposium on Programming*. Springer, March 2012. doi:10.1007/978-3-642-28869-2_4.
- 10 Sebastian Burckhardt, Daan Leijen, Jonathan Protzenko, and Manuel Fähndrich. Global sequence protocol: A robust abstraction for replicated shared state. In *29th European Conference on Object-Oriented Programming, ECOOP 2015, July 5-10, 2015, Prague, Czech Republic*, pages 568–590, 2015. doi:10.4230/LIPIcs.ECOOP.2015.568.
- 11 Andrea Cerone, Giovanni Bernardi, and Alexey Gotsman. A Framework for Transactional Consistency Models with Atomic Visibility. In Luca Aceto and David de Frutos Escrig, editors, *26th International Conference on Concurrency Theory (CONCUR 2015)*, volume 42 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 58–71, Dagstuhl, Germany, 2015. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. doi:10.4230/LIPIcs.CONCUR.2015.58.
- 12 Andrea Cerone and Alexey Gotsman. Analysing Snapshot Isolation. In *Proceedings of the 2016 ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing, PODC’16*, pages 55–64. ACM, 2016. doi:10.1145/2933057.2933096.
- 13 Andrea Cerone, Alexey Gotsman, and Hongseok Yang. Transaction chopping for parallel snapshot isolation. In *Proceedings of the 29th International Symposium on Distributed Computing - Volume 9363, DISC 2015*, page 388–404, Berlin, Heidelberg, 2015. Springer-Verlag. doi:10.1007/978-3-662-48653-5_26.
- 14 Andrea Cerone, Alexey Gotsman, and Hongseok Yang. Algebraic Laws for Weak Consistency. In Roland Meyer and Uwe Nestmann, editors, *Proceedings of the 27th International Conference on Concurrency Theory*, volume 85 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 26:1–26:18, Dagstuhl, Germany, 2017. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. doi:10.4230/LIPIcs.CONCUR.2017.26.

- 15 Nathan Chong, Tyler Sorensen, and John Wickerson. The semantics of transactions and weak memory in x86, power, arm, and C++. In Jeffrey S. Foster and Dan Grossman, editors, *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2018, Philadelphia, PA, USA, June 18-22, 2018*, pages 211–225. ACM, 2018. doi:10.1145/3192366.3192373.
- 16 Natacha Crooks, Youer Pu, Lorenzo Alvisi, and Allen Clement. Seeing is Believing: A Client-Centric Specification of Database Isolation. In *Proceedings of the 2017 ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing, PODC'17*, pages 73–82, New York, NY, USA, 2017. ACM. doi:10.1145/3087801.3087802.
- 17 Pedro da Rocha Pinto, Thomas Dinsdale-Young, and Philippa Gardner. TaDA: A Logic for Time and Data Abstraction. In Richard E. Jones, editor, *Proceedings of the 28th European Conference on Object-Oriented Programming*, volume 8586 of *Lecture Notes in Computer Science*, pages 207–231. Springer, July 2014. doi:10.1007/978-3-662-44202-9_9.
- 18 Khuzaima Daudjee and Kenneth Salem. Lazy database replication with snapshot isolation. In *Proceedings of the 32nd International Conference on Very Large Data Bases, VLDB '06*, page 715–726. VLDB Endowment, 2006. doi:10.5555/1182635.1164189.
- 19 Thomas Dinsdale-Young, Mike Dodds, Philippa Gardner, Matthew J. Parkinson, and Viktor Vafeiadis. Concurrent Abstract Predicates. In *Proceedings of the 24th European Conference on Object-Oriented Programming, ECOOP'10*, pages 504–528. Springer-Verlag, 2010. doi:10.1007/978-3-642-14107-2_24.
- 20 Simon Doherty, Brijesh Dongol, Heike Wehrheim, and John Derrick. Verifying C11 programs operationally. In *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming, PPOPP '19*, pages 355–365, New York, NY, USA, 2019. ACM. doi:10.1145/3293883.3295702.
- 21 Jiaqing Du, Sameh Elnikety, and Willy Zwaenepoel. Clock-SI: Snapshot Isolation for Partitioned Data Stores Using Loosely Synchronized Clocks. In *Proceedings of the 32nd Leibniz International Proceedings in Informatics (LIPIcs), SRDS'13*, pages 173–184, Washington, DC, USA, 2013. IEEE Computer Society. doi:10.1109/SRDS.2013.26.
- 22 Sameh Elnikety, Willy Zwaenepoel, and Fernando Pedone. Database replication using generalized snapshot isolation. In *Proceedings of the 24th IEEE Symposium on Reliable Distributed Systems, SRDS '05*, page 73–84, USA, 2005. IEEE Computer Society. doi:10.1109/RELDIS.2005.14.
- 23 Alan Fekete, Dimitrios Liarokapis, Elizabeth O'Neil, Patrick O'Neil, and Dennis Shasha. Making Snapshot Isolation Serializable. *ACM Transactions on Database Systems*, 30(2):492–528, June 2005. doi:10.1145/1071610.1071615.
- 24 Alexey Gotsman, Hongseok Yang, Carla Ferreira, Mahsa Najafzadeh, and Marc Shapiro. 'cause i'm strong enough: Reasoning about consistency choices in distributed systems. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '16*, page 371–384, New York, NY, USA, 2016. Association for Computing Machinery. doi:10.1145/2837614.2837625.
- 25 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 *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL'15*, pages 637–650. ACM, 2015. doi:10.1145/2676726.2676980.
- 26 Gowtham Kaki, Kapil Earanky, KC Sivaramakrishnan, and Suresh Jagannathan. Safe replication through bounded concurrency verification. *Proc. ACM Program. Lang.*, 2(OOPSLA), October 2018. doi:10.1145/3276534.
- 27 Gowtham Kaki, Kartik Nagar, Mahsa Najafzadeh, and Suresh Jagannathan. Alone Together: Compositional Reasoning and Inference for Weak Isolation. *Proceedings of the ACM on Programming Languages*, 2(POPL):27:1–27:34, December 2017. doi:10.1145/3158115.

- 28 Jeehoon Kang, Chung-Kil Hur, Ori Lahav, Viktor Vafeiadis, and Derek Dreyer. A Promising Semantics for Relaxed-memory Concurrency. In *Proceedings of the 44th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL'17, pages 175–189, New York, NY, USA, 2017. ACM. doi:10.1145/3009837.3009850.
- 29 Eric Koskinen and Matthew Parkinson. The push/pull model of transactions. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '15, page 186–195, New York, NY, USA, 2015. Association for Computing Machinery. doi:10.1145/2737924.2737995.
- 30 Cheng Li, Daniel Porto, Allen Clement, Johannes Gehrke, Nuno Preguiça, and Rodrigo Rodrigues. Making geo-replicated systems fast as possible, consistent when necessary. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI'12, page 265–278, USA, 2012. USENIX Association. URL: <http://www.cs.otago.ac.nz/cosc440/readings/osdi12-final-162.pdf>.
- 31 Richard J. Lipton. Reduction: A method of proving properties of parallel programs. *Commun. ACM*, 18(12):717–721, December 1975. doi:10.1145/361227.361234.
- 32 Si Liu, Peter Csaba Ölveczky, Keshav Santhanam, Qi Wang, Indranil Gupta, and José Meseguer. ROLA: A New Distributed Transaction Protocol and Its Formal Analysis. In Alessandra Russo and Andy Schürr, editors, *Fundamental Approaches to Software Engineering*, pages 77–93, Cham, 2018. Springer. doi:10.1007/978-3-319-89363-1_5.
- 33 Wyatt Lloyd, Michael J. Freedman, Michael Kaminsky, and David G. Andersen. Don't settle for eventual: Scalable causal consistency for wide-area storage with cops. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, page 401–416, New York, NY, USA, 2011. Association for Computing Machinery. doi:10.1145/2043556.2043593.
- 34 Wyatt Lloyd, Michael J. Freedman, Michael Kaminsky, and David G. Andersen. Stronger Semantics for Low-Latency Geo-Replicated Storage. In *Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation*, pages 313–328, Lombard, IL, 2013. USENIX. URL: <https://www.usenix.org/conference/nsdi13/technical-sessions/presentation/lloyd>.
- 35 Kartik Nagar and Suresh Jagannathan. Automated Detection of Serializability Violations Under Weak Consistency. In *Proceedings of the 29th International Conference on Concurrency Theory*, pages 41:1–41:18, 2018. doi:10.4230/LIPIcs.CONCUR.2018.41.
- 36 Aleksandar Nanevski, Yuy Ley-wild, Ilya Sergey, and Germán Andrés Delbianco. *Communicating State Transition Systems for fine-grained concurrent resources*, pages 290–310. Lecture Notes in Computer Science. springer-verlag, 2014. doi:10.1007/978-3-642-54833-8_16.
- 37 Christos H. Papadimitriou. The serializability of concurrent database updates. *J. ACM*, 26(4):631–653, October 1979. doi:10.1145/322154.322158.
- 38 Azalea Raad, Ori Lahav, and Viktor Vafeiadis. On Parallel Snapshot Isolation and Release/Acquire Consistency. In Amal Ahmed, editor, *Proceedings of the 27th European Symposium on Programming*, pages 940–967, Cham, 2018. Lecture Notes in Computer Science. doi:10.1007/978-3-319-89884-1_33.
- 39 The RUBiS benchmark, 2008. URL: <https://rubis.ow2.org/index.html>.
- 40 Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. Conflict-free replicated data types. In *Proceedings of the 13th International Conference on Stabilization, Safety, and Security of Distributed Systems*, SSS'11, page 386–400, Berlin, Heidelberg, 2011. Springer-Verlag. doi:10.1007/978-3-642-24550-3_29.
- 41 Dennis Shasha, Francois Llirbat, Eric Simon, and Patrick Valduriez. Transaction Chopping: Algorithms and Performance Studies. *ACM Transactions on Database Systems*, 20(3):325–363, September 1995. doi:10.1145/211414.211427.
- 42 Yair Sovran, Russell Power, Marcos K. Aguilera, and Jinyang Li. Transactional Storage for Geo-replicated Systems. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles*, SOSP'11, pages 385–400, New York, NY, USA, 2011. ACM. doi:10.1145/2043556.2043592.

- 43 Kristina Spirovska, Diego Didona, and Willy Zwaenepoel. Wren: Nonblocking Reads in a Partitioned Transactional Causally Consistent Data Store. In *Proceedings of the 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN'18*, pages 1–12, 2018. doi:10.1109/DSN.2018.00014.
- 44 The TPC-C benchmark, 1992. URL: <http://www.tpc.org/tpcc/>.
- 45 Werner Vogels. Eventually consistent. *Commun. ACM*, 52(1):40–44, January 2009. doi:10.1145/1435417.1435432.
- 46 Shale Xiong. *Parametric Operational Semantics for Consistency Models*. PhD thesis, Imperial College London, April 2021. URL: <http://www.shalexiong.com/thesis.pdf>.
- 47 Peter Zeller. Testing properties of weakly consistent programs with repliss. In *Proceedings of the 3rd International Workshop on Principles and Practice of Consistency for Distributed Data, PaPoC'17*, pages 3:1–3:5, New York, NY, USA, 2017. ACM. doi:10.1145/3064889.3064893.

Putting Randomized Compiler Testing into Production

Alastair F. Donaldson 

Google, London, United Kingdom
Imperial College London, United Kingdom
afdx@google.com

Hugues Evrard

Google, London, United Kingdom
hevrard@google.com

Paul Thomson

Google, London, United Kingdom
paulthomson@google.com

Abstract

We describe our experience over the last 18 months on a compiler testing technology transfer project: taking the GraphicsFuzz research project on randomized metamorphic testing of graphics shader compilers, and building the necessary tooling around it to provide a highly automated process for improving the Khronos Vulkan Conformance Test Suite (CTS) with test cases that expose fuzzer-found compiler bugs, or that plug gaps in test coverage. We present this tooling for test automation – `gfauto` – in detail, as well as our use of differential coverage and test case reduction as a method for automatically synthesizing tests that fill coverage gaps. We explain the value that GraphicsFuzz has provided in automatically testing the ecosystem of tools for transforming, optimizing and validating Vulkan shaders, and the challenges faced when testing a tool ecosystem rather than a single tool. We discuss practical issues associated with putting automated metamorphic testing into production, related to test case validity, bug de-duplication and floating-point precision, and provide illustrative examples of bugs found during our work.

2012 ACM Subject Classification Software and its engineering → Compilers; Software and its engineering → Software testing and debugging

Keywords and phrases Compilers, metamorphic testing, 3D graphics, experience report

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2020.22

Category Experience Report

Supplementary Material ECOOP 2020 Artifact Evaluation approved artifact available at <https://doi.org/10.4230/DARTS.6.2.3>.

Acknowledgements We are grateful to David Neto and to the anonymous ECOOP 2020 reviewers for their feedback on an earlier draft of this work.

1 Introduction

Graphics processing units (GPUs) provide hardware-accelerated graphics in many scenarios, such as 3D and 2D games, applications, web browsers, and operating system user interfaces. To utilize GPUs, developers must use a graphics programming API, such as OpenGL, Direct3D, Metal or Vulkan, and write *shader programs* that execute on the GPU in an embarrassingly-parallel manner. Shaders are written in a *shading language* such as GLSL, HLSL, MetalSL, or SPIR-V (associated with the OpenGL, Direct3D, Metal and Vulkan APIs, respectively), and are usually portable enough to run on many different GPU models. A graphics driver contains one or more *shader compilers* to translate shaders from portable shading languages to machine code specific to the system's GPU.



© Alastair F. Donaldson, Hugues Evrard, and Paul Thomson;
licensed under Creative Commons License CC-BY
34th European Conference on Object-Oriented Programming (ECOOP 2020).

Editors: Robert Hirschfeld and Tobias Pape; Article No. 22; pp. 22:1–22:29

Leibniz International Proceedings in Informatics



LIPICs Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany



Functional defects in graphics shader compilers can have serious consequences. Clearly, as with any bug in any application, it is undesirable if a mis-compiled graphics shader causes unintended visual effects. Furthermore, since shaders are compiled at *runtime* (because the GPU and driver that will be present when an application executes is not known at the application's compile time), a shader compiler crash can lead to an overall application crash. Additionally, developers cannot feasibly test for or workaround these issues, as the driver version that crashes may not have even been written at the time of application development. Worse still, because the graphics driver usually drives the whole system's display, if a shader compiler defect leads to the state of the driver being corrupted, the entire system may become unstable. This can lead to device freezes and reboots, display corruption and information leakage; see [15] for a discussion of some examples, including information leak bugs in iOS [2] (CVE-2017-2424) and Chrome [5] caused by GPU driver bugs, and an NVIDIA machine freeze [38] (CVE-2017-6259).

One way to provide a degree of graphics driver quality – and shader compiler quality in particular – is via standardized test suites. An example is the Khronos Vulkan Conformance Test Suite (Vulkan CTS, or just CTS for short) [25]. This is a large set of functional tests for implementations of the Vulkan graphics API. The Khronos Group, who define various standards and APIs including Vulkan, requires a Vulkan implementation (such as a GPU driver and its associated GPU hardware) to demonstrate that they pass the Vulkan CTS tests in order for the vendor to use the official Vulkan branding. Google's Android Compatibility Test Suite incorporates the Vulkan CTS, so that Android devices that provide Vulkan capabilities must include drivers that pass the Vulkan CTS. Improving the quality and thoroughness of the Vulkan CTS is thus an indirect method for improving the quality of Vulkan graphics drivers in general, and on Android in particular.

GraphicsFuzz (originally called **GLFuzz**) [16, 15] is a technique and tool chain for automatically finding crash and miscompilation bugs in shader compilers using metamorphic testing [9, 42]. Whenever **GraphicsFuzz** synthesizes a test that exposes a bug in a conformant Vulkan driver, this demonstrates a gap in the Vulkan CTS: the driver has passed the conformance test suite despite exhibiting this bug. If **GraphicsFuzz** synthesizes a test that covers a part of a conformant driver's source code, but the driver does not crash, and the code is not covered by any existing CTS tests, then this also exposes a CTS gap (albeit arguably a less severe one): it demonstrates that part of the driver's source code *can* be covered but is *not* covered by the CTS; bugs that creep into such code in the future would not be caught.

In this experience report we describe our activities at Google over the last 18 months putting the **GraphicsFuzz** tool chain into production, with the aim of improving implementations of the Vulkan API. We have set up a process whereby the randomized metamorphic testing capabilities of **GraphicsFuzz** are used to find tests that expose driver bugs or CTS coverage gaps, shrink such tests down to small examples that are simple enough for humans to read and debug, and package the resulting tests into a form whereby they are almost completely ready to be added to the Vulkan CTS. So far, this has led to 122 tests that exposed driver and tooling bugs and 113 that exposed driver coverage gaps being added to CTS. The bugs affect a range of mobile and desktop drivers, as well as tools in the SPIR-V ecosystem. Our contribution of CTS tests that expose them means that future conformant Vulkan drivers cannot exhibit them (at least not in a form that causes these tests to fail).

We start by presenting relevant background on graphics programming APIs, shader processing tools, the Vulkan CTS, and the **GraphicsFuzz** testing approach (§2). We then describe how we set up a pathway for incorporating tests that expose bugs found by **GraphicsFuzz** into the CTS, and various practical issues we had to solve to ensure valid

tests (§3). With this pathway in place we were empowered to build a fuzzing framework, `gfauto`, for running `GraphicsFuzz` against a range of drivers and shader processing tools, automatically shrinking tests that find bugs and getting them into a near-CTS-ready form (§4). To aid in finding coverage gaps, we have built tooling for *differential* coverage analysis; we describe how – by treating coverage gaps as bugs – `gfauto` can be used to synthesize tests that expose such gaps in a highly automatic fashion (§5). A strength of `GraphicsFuzz` is that it facilitates testing not only vendor graphics drivers, but also a variety of translation, optimization and validation tools that are part of the Vulkan ecosystem. We explain how this also presents a challenge: it can be difficult to determine which component of the ecosystem is responsible for a bug (§6). Throughout, we provide illustrative examples of noteworthy bugs and tests found and generated by our approach, including bugs that affect core infrastructure (such as LLVM), bugs that affect multiple tools simultaneously, and bugs for which the responsible tool is non-trivial to identify. We conclude by discussing related (§7) and future (§8) work.

Main takeaways. We hope this report is simply interesting for researchers and practitioners to read as an example of successful technology transfer of research ideas to industrial practice. In addition, we believe the following aspects could provide inspiration for follow-on research:

- The pros and cons of fuzzing a low level language via a program generator for a higher level language and a suite of translation and optimization tools, including the problem of how to determine *where* in a tool chain a fault has occurred (§3.1 and §6);
- The need for image differencing algorithms that are well-suited to tolerating the degree of variation we expect from graphics shaders due to floating-point precision (§3.4);
- Threats to test validity caused by undefined behavior, long-running loops and floating-point precision, where more advanced program analyses have the potential to be applied (§3.5);
- The difficulty of correctly maintaining a test case generator and a corresponding test case reducer, especially when test case reduction needs to be semantics-preserving (also §3.5)
- The challenge of de-duplicating bugs that do not exhibit distinguished characteristics, such as wrong image bugs and message-free compile and link errors (§4.2);
- The idea of using differential coverage analysis and test case reduction to fill coverage gaps (§5), and the challenge of going beyond synthesizing tests that trivially cover new code to tests that are also equipped with meaningful oracles (§5.4 specifically).

Open sourcing. Our extensions to the `GraphicsFuzz` tool, the new `gfauto` tool, and our infrastructure for differential code coverage, are open source.¹ The tests we have contributed to Vulkan CTS are also open source.²

2 Background

2.1 The GLSL and SPIR-V Shading Languages

GLSL. The OpenGL Shading Language (GLSL) [22] is the main shading language in the OpenGL graphics API [41] (analogous to HLSL and the Direct3D API). It is used for rendering hardware-accelerated 2D and 3D graphics. OpenGL ES [32], and its associated

¹ <https://github.com/google/graphicsfuzz>

² <https://github.com/KhronosGroup/VK-GL-CTS/tree/master/external/vulkancts/data/vulkan/amber/graphicsfuzz>

22:4 Putting Randomized Compiler Testing into Production

```
1 precision highp float;
2
3 layout(location = 0) out vec4 _GLF_color;
4
5 void main()
6 {
7     vec2 a = vec2(1.0);
8     vec4 b = vec4(1.0);
9     pow(vec4(a, vec2(1.0)), b);
10    // Added manually to ensure that the shader writes red
11    _GLF_color = vec4(1.0, 0.0, 0.0, 1.0);
12 }
```

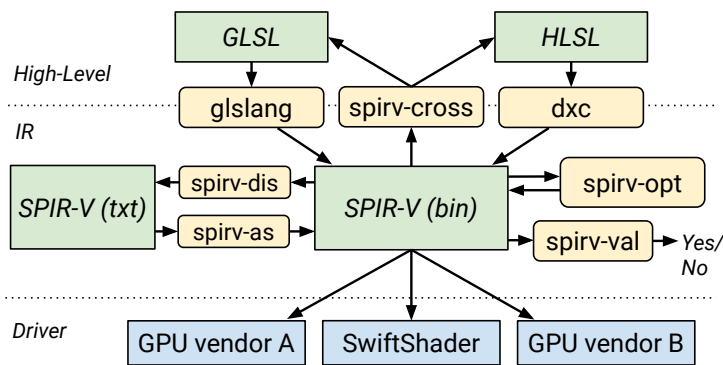
■ **Figure 1** A reduced GLSL ES fragment shader that, after translation to SPIR-V, triggered a bug leading to a crash in the GPU shader compiler for a popular Android device.

shading language GLSL ES [43], is a subset of the OpenGL API supported by mobile devices, including Android devices.³ We focus on the GLSL ES shading language version 3.10 and later, and henceforth drop the ES suffix and version number for brevity. Figure 1 shows an example of a GLSL *fragment* shader (also known as a pixel shader in Direct3D). The code is C-like, but with some additional features useful for graphics programming. The code in `main` is conceptually executed n times on the GPU for each of the n pixels rendered into a *framebuffer* (which stores the image) using the shader. The `precision highp float;` line causes all subsequent floating-point values to be represented with 32 bits of precision by default (lower precision can be specified via the `mediump` and `lowp` qualifiers on a per-variable basis). Note that `main` has no parameters and a `void` return type; in GLSL, inputs and outputs to the shader are instead expressed using special global variables. Global variable `_GLF_color`⁴ is an output variable into which the fragment shader writes the RGBA colour value that will be rendered at the pixel coordinate for which the shader is running. The `vec2` and `vec4` types are built-in float vector types (with 2 and 4 float components respectively). The vector constructor form that takes one floating-point value (e.g. `vec2(1.0)`) creates a vector with all components set to that value. A vector constructor can also take a combination of vectors and/or scalars (e.g. `vec4(a, vec2(1.0))`) to construct a vector made up of each component in order, as long as the total number of components matches the vector type. The `pow(x, y)` function yields an approximation of x^y , and is an example of one of the many built-in math functions provided in GLSL. The example of Figure 1 was minimized with the aim of reproducing a shader compiler crash bug (discussed further as Example 1 below), and is not representative of a practically useful graphics shader: the `main` function performs some redundant computation and then writes the colour red (`vec4(1.0, 0.0, 0.0, 1.0)`) to the output colour variable. Thus, every pixel rendered by this shader will be red.

Shaders can also define *uniform* global variables (not shown in the example), using the `uniform` keyword. These are shader inputs that yield the same value for every pixel being shaded during a single shader invocation. For example, a uniform declaration `uniform float time;` could be used to pass a representation of the current time into a shader, allowing it to produce a time-varying visual effect.

³ Strictly, OpenGL ES is not quite a subset of OpenGL: over time it has evolved with some features that have been deemed specifically important for mobile platforms.

⁴ The `_GLF` prefix comes from the fact that the tool was originally called `GLFuzz`. This prefix is used as a default for any special variable, function or macro names used by `GraphicsFuzz`.



■ **Figure 2** Diagram showing the various tools in the SPIR-V ecosystem and how they interact.

SPIR-V. In comparison to OpenGL, Vulkan [20] is a newer, lower-level graphics API. It is widely supported by modern desktop GPUs, as well as being available on newer Android devices. Standard, Portable Intermediate Representation - V (SPIR-V; the “V” does not stand for anything) is the Vulkan shading language [23]. Unlike GLSL, SPIR-V was designed as an intermediate representation to be stored in a binary form, and thus is not usually written directly by programmers. Instead, programmers write their shaders in a higher-level language like GLSL or HLSL, and use a tool to compile the shaders into SPIR-V. SPIR-V modules use static single assignment (SSA) form [12], including the use of Phi instructions [12], and functions contain blocks with branches.

2.2 The SPIR-V Tooling Ecosystem

Figure 2 summarizes various open source tools for analyzing and transforming SPIR-V shaders, and translating to and from SPIR-V.

As mentioned in §2.1, most shaders are written in high level languages such as GLSL and HLSL and translated to SPIR-V. For example, `glslang` [24] and `DXC` [37] can compile GLSL and HLSL, respectively, to SPIR-V. A binary SPIR-V shader can be loaded by the Vulkan API and executed as part of a graphics pipeline on a GPU device. Google provides a software implementation of Vulkan, `SwiftShader` [18], which allows Vulkan applications (including their SPIR-V shaders) to be executed in the absence of Vulkan-capable hardware. This is useful to bring Vulkan support to old devices, as a fall-back renderer if a GPU driver goes into an unstable state, and as a “second opinion” for GPU driver writers.

The code generated by front-ends such as `glslang` and `DXC` is not typically optimized. In fact `glslang` deliberately performs as straightforward a syntax-directed translation of a GLSL shader as possible. The `spirv-opt` tool, part of the Khronos SPIRV-Tools framework [27], implements many target-agnostic optimizations as SPIR-V-to-SPIR-V passes.

The philosophy of the Vulkan API is to allow drivers to assume that the Vulkan workloads with which they are presented are valid, pushing the onus of validation to the application. In support of this, the `spirv-val` tool (also part of the SPIR-V tools framework), checks whether a SPIR-V shader obeys the (many) rules mandated by the SPIR-V specification [23]. The `spirv-dis` and `spirv-as` disassembler and assembler (again, part of SPIRV-Tools) allow a shader to be translated into text format and back, which is useful for debugging.

Finally, the `spirv-cross` tool [28] allows SPIR-V to be translated into various shading languages including GLSL, HLSL and Apple’s Metal shading language (MetalSL, not shown in the figure). Translation to these higher-level languages can help in understanding the

intended behavior of a SPIR-V shader, and the SPIR-V-to-MetalSL pathway is used by the MoltenVK project, which provides an implementation of most of Vulkan on top of Apple's Metal graphics API [26].

2.3 The Vulkan Conformance Test Suite

The Khronos Vulkan Conformance Test Suite (Vulkan CTS) [25] is a set of tests for the Vulkan API. In theory, every part of the Vulkan specification should have one or more corresponding tests in the Vulkan CTS. Each test should invoke the relevant Vulkan API functions to check that a Vulkan implementation conforms to the Vulkan specification. Indeed, the Vulkan CTS mostly consists of a set of functional tests (there are over 550,000 Vulkan CTS tests at the time of writing) that attempt to test features in isolation. The Vulkan CTS is part of the larger Khronos Conformance Test Suite called dEQP (drawElements Quality Program⁵) that additionally contains tests for OpenGL ES and EGL.

Any implementation of Vulkan (including any Vulkan graphics driver with its associated GPUs) must pass the Vulkan CTS (and upload the results to Khronos for peer review) before the Vulkan name or logo can be used in association with the implementation. Thus, the Vulkan CTS sets a minimum quality standard for every conformant Vulkan implementation. Of course, the test suite is also extremely useful during development of a Vulkan driver; as with most test suites, it can be used to identify bugs and regressions, and to measure progress towards becoming a conformant implementation. The OpenGL ES and EGL test suite is similarly used as part of the conformance process for those APIs, and as a useful aid during driver development. The dEQP test suite is included in the Android Compatibility Test Suite (Android CTS), which is an even larger test suite for Android devices. Original equipment manufacturers (OEMs) will typically customize the Android OS for a given device, but these Android implementations must still pass the Android CTS to be deemed “compatible”. Thus, the Vulkan CTS also sets a minimum quality standard for Vulkan on every compatible Android device, which can have a large impact on the Android ecosystem.

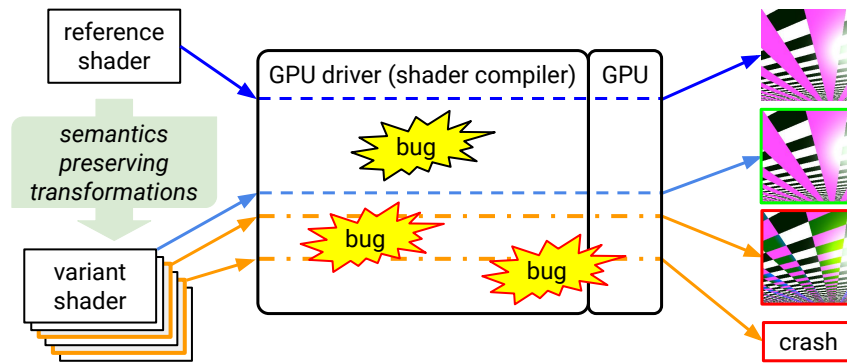
Vulkan CTS development is mostly done by Khronos members, although anybody can contribute. New tests are reviewed by GPU vendors before being accepted. Tests need to be deterministic, and clear enough to allow debugging of Vulkan implementations if a test fails.

2.4 Metamorphic Compiler Testing Using GraphicsFuzz

The GraphicsFuzz tool originated from a research project at Imperial College London, and formed the basis of a spin-out company, GraphicsFuzz Ltd., founded by the authors of this paper, which Google acquired during 2018.

Figure 3 gives an overview of the GraphicsFuzz approach to testing shader compilers. It starts with an existing reference shader which, after being compiled by the shader compiler embedded in the GPU driver and executed on the GPU hardware, leads to a given reference image. A classic way to test a GPU driver would be to compare this resulting image to what a reference implementation of the graphics API would produce. However, graphics API are purposefully relaxed to let GPU vendors reach very high performance through aggressive optimizations, such that there are various images that can be deemed acceptable. It is currently not possible, and not desirable for GPU vendors, to agree on a strict reference implementation that would serve as a test oracle.

⁵ dEQP was a commercial product developed by the drawElements company. Google acquired drawElements in 2014 and donated the dEQP test suite to Khronos, where it is now open source.



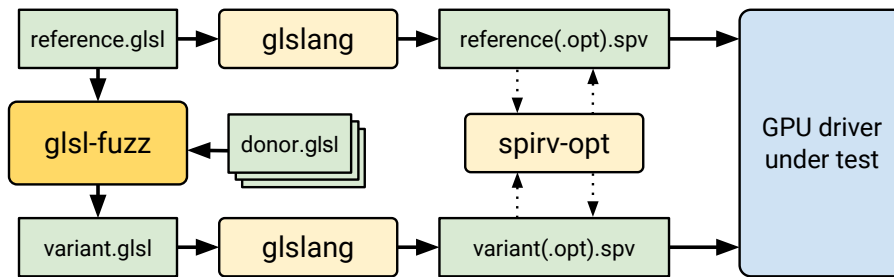
■ **Figure 3** Illustration of the metamorphic testing approach used by GraphicsFuzz.

Inspired by the *equivalence modulo inputs* method for testing C compilers [31], GraphicsFuzz works around this lack of oracle by using *metamorphic testing* [9, 42]: here the GPU input (shader) is transformed in a way that should not change its output (image). In practice, the `gsl-fuzz` tool applies *semantics-preserving transformations* to the reference shader source code to obtain a family of variant shaders. As a very simple example, one can add zero to an existing integer operation, `int x = y + 0`: this source code change should not impact the program behavior. The `gsl-fuzz` tool contains many such semantics-preserving transformations [15], including: arithmetic operations (such as adding zero, multiplying by one, etc), boolean operations (e.g. `bool b = x && true`), dead code injection (adding valid yet unreachable code, e.g. wrapped inside an `if (false) { ... }`), live code injection (adding code that will be executed while making sure to save and restore all variables affected by it, e.g. `{t = x; x = foo(x); x = t;}`), control flow wrapping (e.g. wrapping existing code in a single-iteration loop `do { ... } while(false)`), packing scalar data into composite data types (such as structs and vectors), and outlining expressions into functions. Some of the values used in these transformations, such as zero, one, true and false, are obtained via program inputs whose value is guaranteed at execution time, but unknown at compilation time. This is to make sure compilers cannot trivially remove some transformations, e.g. by statically detecting dead code injections to be unreachable.

Care is required when applying these transformations to ensure that program semantics are preserved. For instance, one cannot wrap some code in a single-iteration loop if this code contains a top-level `break` statement: the `break` would now apply to the newly-introduced loop, rather than to the original loop or switch statement in which it originally appeared.

Each variant shader is syntactically distinct from the reference, yet has the same semantics (modulo floating-point error). It may thus exercise a different path in the shader compiler but should still lead to a visually similar image being rendered, so long as the reference shader is sufficiently numerically stable. This is illustrated by the top, blue variant shader line in Figure 3. However, some variants may lead to significantly different images, or a driver crash, which are symptoms of bugs, most likely in the shader compiler but also potentially in other parts of the driver or GPU hardware. These are illustrated by the lower two orange variant shader lines in Figure 3. For a given GPU, we cannot know what to expect as a reference image, but we do expect variants to lead to an extremely similar image.

Semantics-preserving transformations are used in other contexts, e.g. compiler optimizations and code obfuscation tools modifying a program representation while keeping the same behavior. Code refactoring, when understood as improving the program structure while keeping the same functional features, can also be considered as a semantics-preserving



■ **Figure 4** Targeting SPIR-V shader compilers from GLSL.

transformation at a bigger scale than the transformations used in `glsl-fuzz`. For our testing purpose, we are interested in any kind of semantics-preserving transformation that may potentially have interesting effects on how the shader is processed by the GPU.

Although a bug-inducing variant can be used as a starting point for debugging, its source code is often barely understandable by a human because of the hundreds of transformations that have been applied to it. To ease debugging, the `glsl-reduce` tool progressively shrinks the variant source code while making sure that the bug is still triggered.

There are two reduction modes:

Semantics-preserving reduction. For shader miscompilation bugs leading to wrong images, `glsl-reduce` performs *semantics-preserving reduction* by removing the `glsl-fuzz` transformations in a way that still preserves semantics. This typically leads to a variant that differs from its reference only by a handful of transformations necessary to trigger the wrong image bug. The pair of semantically identical shaders is useful as a debugging starting point.

Semantics-changing reduction. For bugs leading to a driver crash, `glsl-reduce` performs *semantics-changing reduction* by removing source code, the only requirement being to keep it statically valid (which includes being syntactically valid and well-typed). No valid shader should not cause a driver crash, so there is no need to keep a semantic equivalence with the reference shader. Semantics-changing reductions can lead to very short crash-inducing shaders (e.g. Figure 1), which are useful for debugging and as regression test cases.

3 Integrating GraphicsFuzz Tests With Vulkan CTS

As described in §2.4, the `GraphicsFuzz` tool was originally designed to find bugs in OpenGL and OpenGL ES drivers by transforming shaders written in the GLSL shading language. However, our interest is in making shader compilers for the more modern Vulkan API as reliable as possible by improving the Vulkan CTS, and Vulkan uses SPIR-V as its shading language (see §2.1). We explain the process we used to allow GLSL-based fuzzing of SPIR-V shader compilers via translation (§3.1). We explain why we did *not* opt for embedding the fuzzer inside CTS, or directly contributing large numbers of fuzzer-generated tests, instead preferring to add tests that are known to expose shader compiler bugs (§3.2). We then describe how we paved the way for tests that expose crash and wrong image bugs to be added to CTS (§3.3 and §3.4, respectively).

3.1 Fuzzing SPIR-V Compilers via GLSL Shaders

In order to target SPIR-V shader compilers with a tool that operates on GLSL, we leverage the `glslang` translator, which takes GLSL as input and has a SPIR-V back-end. By design, `glslang` performs a very straightforward translation from GLSL to SPIR-V, performing no optimization beyond some basic constant folding and elimination of functions that are never invoked. As a result, the SPIR-V that `glslang` emits is rather basic (e.g. it rarely exhibits uses of Phi instructions). While it is vital that SPIR-V shader compilers correctly handle this “vanilla” SPIR-V, we are also interested in testing their support for more interesting SPIR-V features. Towards this aim, we optionally invoke the `spirv-opt` tool on the SPIR-V that `glslang` generates, with its `-O` flag (optimize for speed), its `-Os` flag (optimize for size), or a random selection of its finer-grained flags (which include things like `--ssa-rewrite`, which changes variable uses to register uses, and can add Phi instructions, and `--eliminate-dead-inserts`, which avoids unnecessary insertions of data into composite structures).

This use of `glslang` and `spirv-opt` allows us to perform metamorphic fuzzing at the GLSL level to generate a variant from a reference, send both the variant and reference through `glslang` to turn them into SPIR-V, and then (optionally, and at random) transform the variant using a configuration of `spirv-opt`. The resulting SPIR-V shaders can then be compiled and executed on a Vulkan driver, and the results they compute can be compared. This process is illustrated graphically in Figure 4. This translation-based approach allows us to also find bugs in `glslang` and `spirv-opt`, which benefits the Vulkan ecosystem. However, as discussed further in §6, it can be hard to determine – in the case of wrong image bugs – which of these tools or the driver’s shader compiler has miscompiled.

Making shaders “Vulkan-friendly”. Unlike in GLSL, where a global variable of almost any type can be declared as `uniform` (see §2.1), SPIR-V requires that every uniform is declared as a field of a structure called a *uniform block*, with the whole structure being declared to be `uniform`. The number of uniform blocks allowed in a SPIR-V module is implementation-dependent. GLSL has been updated with “Vulkan-friendly features” to allow uniforms to be presented in this way, and `glslang` will only compile Vulkan-friendly shaders into SPIR-V. We thus wrote a simple pass to turn a standard GLSL shader into Vulkan-friendly form. For simplicity of implementation we approached this by placing each original uniform variable in its own (single-field) uniform block. Our pass limits the number of such blocks to 10, as we have not encountered a Vulkan implementation that supports fewer than 10 uniform blocks, and none of the reference shaders we currently use for testing feature more than 10 uniforms. When `glsl-fuzz` generates a variant shader with more than 10 uniforms (due to injecting code from other shaders), our Vulkan preparation pass demotes superfluous uniforms to standard global variables initialized to concrete values.

3.2 Argument for Not Running Fuzzing in CTS

We briefly considered pitching to Khronos the idea of running `GraphicsFuzz` as part of running CTS, so that to pass CTS a driver would have to pass all of the regular tests, and additionally survive running a certain number of `GraphicsFuzz`-generated tests unscathed. We quickly dismissed this idea because it is important to GPU driver makers that qualifying as Vulkan-conformant involves passing a fixed number of tests that run in a deterministic fashion. However much enthusiasm driver makers have for randomized testing as a way to discover bugs, it is understandable that there is little appetite for a conformance test suite that exhibits randomization.

Another issue with embedding GraphicsFuzz in CTS is that inevitable defects in GraphicsFuzz (such as generating a variant shader that turns out *not* to be semantically equivalent to the reference shader) would manifest as a driver failing to pass CTS.

An alternative to actually running GraphicsFuzz in CTS would be to generate a reasonably large set of shaders – e.g. 1000 shaders – and contribute them as CTS tests. We also quickly decided against this strategy for a few reasons. First, the intended behavior of a CTS test should be feasible for a Vulkan expert to understand. The generated variant shaders are large (in order to maximize the probability of finding a bug), and not feasible for humans to realistically comprehend in isolation; the reducer, `gsl-reduce`, is essential in shrinking a bug-inducing variant to a comprehensible form. Furthermore, 1000 large randomized shaders would be a substantial addition to CTS in terms of the test suite’s runtime, but is not a large enough number of tests to run with the expectation of thoroughly testing a shader compiler.

We opted instead for setting up a continuous fuzzing process whereby we could use GraphicsFuzz to find bugs that affect current shader compilers, use `gsl-reduce` to shrink the associated tests down to small examples that reproduce said bugs, and contribute the resulting tests. We now explain the format we settled on for adding crash and wrong image tests to CTS. We detail our tooling for continuous fuzzing in §4.

3.3 Supporting Crash Tests

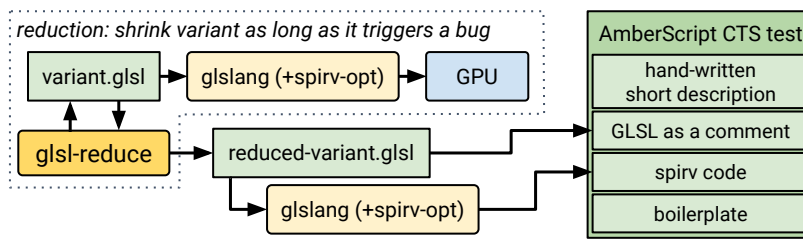
Around the same time we commenced our plan to add tests exposing shader compiler crash bugs to CTS, a new tool called Amber was launched [17]. Amber provides a simple domain-specific language, AmberScript, in which some aspects of a Vulkan graphics pipeline can be specified, including the SPIR-V shaders that should be executed, and input and output data (including uniform blocks and framebuffers) on which the shaders should operate. It also allows querying the results of running a shader, e.g. probing pixels in the output framebuffer. The motivation for Amber was to make it easy to write stand-alone shader compiler tests, hiding the (very substantial amount of) Vulkan API boilerplate required for even a simple graphics pipeline. Since early 2019, Amber has been integrated into Vulkan CTS and is now the preferred method for writing shader compiler tests.

We wrote a script that takes a reduced GLSL shader known to trigger a SPIR-V shader compiler crash (after translation to SPIR-V and possibly optimization using some specific `spirv-opt` flags) and produces an Amber test comprised of:

- A brief comment, supplied as an argument to the script, to describe the test and the reason why it should be expected to pass;
- A comment showing the original GLSL code for the reduced shader; this is useful because GLSL is much easier to read compared with SPIR-V;
- Assembly code for the SPIR-V fragment shader that was obtained from this GLSL by translation using `gslang` and (optional) optimization using `spirv-opt`;
- A comment listing the `spirv-opt` arguments that were applied (if any);
- Commands to create the target framebuffer and to populate the shader uniforms;
- A command, supplied as an argument to the script, to check some property of the image finally obtained in the framebuffer.

Figure 5 illustrates the process of Amber test creation following test case reduction.

► **Example 1.** The GLSL shader of Figure 1, which we used to illustrate the GLSL language in §2.1, triggered a SPIR-V shader compiler crash in the GPU driver of a popular Android device, after translation to SPIR-V (and without requiring any use of `spirv-opt`). This shader was reduced from a much larger variant shader generated by GraphicsFuzz, which we edited



■ **Figure 5** Overview of the reduction process and the creation of a CTS test in AmberScript.

by making the variable names simpler, and by adding the final line of executable code, which ensures that the colour red is written to the framebuffer. We believe the shader compiler crash was due to an assertion failing in the lowering of the `pow` intrinsic to LLVM bytecode. This is somewhat surprising given that the result of `pow` is not used, but was presumably due to dead code elimination being executed after lowering.

An abbreviated version of the Amber test corresponding to this shader is shown in Figure 6 (we omit some of the SPIR-V assembly). The test and its intent are described on lines 1–6; line 8 indicates that a standard trivial *vertex* shader (not otherwise relevant in this experience report) should be used in the test pipeline; lines 10–24 show the GLSL code for the fragment shader, and match Figure 1; the corresponding SPIR-V shader (emitted by `glslang`) is shown on lines 26–52 as SPIR-V assembly (notice the invocation of the `Pow` intrinsic on line 49); line 55 declares a framebuffer, and lines 57–62 define a graphics pipeline based on the vertex and fragment shaders, with the framebuffer attached; line 63 sets the back buffer to black (so that any pixels not rendered to would remain black); lines 65–66 run the pipeline, and line 68 asserts that the framebuffer ends up red at every pixel.

The purpose of adding this test to CTS was to expose the driver bug that it triggered, so that future drivers cannot be Vulkan conformant unless the underlying bug is (at least partially) fixed. We write red to the framebuffer and assert that the framebuffer indeed ends up being red so that the test has at least some runtime oracle; it does a little more than just checking that shader compilation succeeds. The test would be better if the shader stored values into one or more components of `_GLF_color` using the result of the call to `pow`, and then asserted a suitable framebuffer colour; as it stands the test would pass even if `pow` were compiled incorrectly but without a compiler crash. We occasionally work to contribute higher quality test oracles, but do not agonize over this since the main motivation for adding these tests is to force the elimination of compiler crash bugs from conformant drivers.

► **Example 2.** Figure 7 shows a reduced shader that triggered a bug in AMD’s LLVM-Based Pipeline Compiler (LLPC) [19]: an assertion failed during constant folding:

```
amdllpc: external/llvm/lib/Support/APFloat.cpp:1521: llvm::lostFraction llvm::detail::IEEEFloat::
addOrSubtractSignificand(const llvm::detail::IEEEFloat &, bool): Assertion '!carry' failed.
```

We reported this bug,⁶ and the LLPC compiler developers traced its root cause to a bug in LLVM’s floating-point emulation code related to handling of subnormal numbers, which was promptly fixed.⁷ This demonstrates that shader compiler fuzzing can have positive impact on common infrastructure (LLVM in this case) that is used by many compilers for C-family languages. We contributed a Vulkan CTS test based on this bug, with a structure similar to the example of Figure 6.⁸

⁶ <https://github.com/GPUOpen-Drivers/llpc/issues/211>

⁷ <https://reviews.llvm.org/D69772>

⁸ <https://github.com/KhronosGroup/VK-GL-CTS/blob/master/external/vulkancts/data/vulkan/amber/graphicsfuzz/mix-floor-add.amber>

22:12 Putting Randomized Compiler Testing into Production

```
1 # A test for a bug found by GraphicsFuzz.
2
3 # Short description: A fragment shader that uses pow
4
5 # We check that all pixels are red. The test passes because main does
6 # some computation and then writes red to _GLF_color.
7
8 SHADER vertex variant_vertex_shader PASSTHROUGH
9
10 # variant_fragment_shader is derived from the following GLSL:
11 # #version 310 es
12 #
13 # precision highp float;
14 # precision highp int;
15 #
16 # layout(location = 0) out vec4 _GLF_color;
17 #
18 # void main()
19 # {
20 #   vec2 a = vec2(1.0);
21 #   vec4 b = vec4(1.0);
22 #   pow(vec4(a, vec2(1.0)), b);
23 #   _GLF_color = vec4(1.0, 0.0, 0.0, 1.0);
24 # }
25 SHADER fragment variant_fragment_shader SPIRV-ASM
26 ; SPIR-V
27 ; Version: 1.0
28 ; Generator: Khronos Glslang Reference Front End; 7
29 ; Bound: 28
30 ; Schema: 0
31
32     OpCapability Shader
33     %1 = OpExtInstImport "GLSL.std.450"
34     OpMemoryModel Logical GLSL450
35     OpEntryPoint Fragment %main "main" %_GLF_color
36     OpExecutionMode %main OriginUpperLeft
37     OpSource ESSL 310
38     OpName %main "main"
39     OpName %a "a"
40     OpName %b "b"
41     OpName %_GLF_color "_GLF_color"
42     OpDecorate %_GLF_color Location 0
43
44     %void = OpTypeVoid
45     %3 = OpTypeFunction %void
46     %float = OpTypeFloat 32
47     %v2float = OpTypeVector %float 2
48     ...
49     %21 = OpCompositeConstruct %v4float %17 %18 %19 %20
50     %22 = OpLoad %v4float %b
51     %23 = OpExtInst %v4float %1 Pow %21 %22
52     OpStore %_GLF_color %27
53     OpReturn
54     OpFunctionEnd
55
56 END
57
58 BUFFER variant_framebuffer FORMAT B8G8R8A8_UNORM
59
60 PIPELINE graphics variant_pipeline
61   ATTACH variant_vertex_shader
62   ATTACH variant_fragment_shader
63   FRAMEBUFFER_SIZE 256 256
64   BIND BUFFER variant_framebuffer AS color LOCATION 0
65 END
66 CLEAR_COLOR variant_pipeline 0 0 0 255
67
68 CLEAR variant_pipeline
69 RUN variant_pipeline DRAW_RECT POS 0 0 SIZE 256 256
70
71 EXPECT variant_framebuffer IDX 0 0 SIZE 256 256 EQ_RGBA 255 0 0 255
```

■ **Figure 6** CTS test that exposed a shader compiler crash bug, in AmberScript form. Some of the SPIR-V assembly has been omitted.


```
1 #version 310 es
2 precision highp float;
3
4 layout(location = 0) out vec4 _GLF_color;
5
6 vec3 GLF_live6mand()
7 {
8     return mix(uintBitsToFloat(uvec3(38730u, 63193u, 63173u)),
9               floor(vec3(463.499, 4.7, 0.7)), vec3(1.0) + vec3(1.0));
10 }
11 void main()
12 {
13     GLF_live6mand();
14     _GLF_color = vec4(1.0, 0.0, 0.0, 1.0);
15 }
```

■ **Figure 7** Reduced shader that triggered a floating-point constant folding bug in LLVM.

3.4 Supporting Wrong Image Tests

Recall from §2.4 and Figure 3 that GraphicsFuzz finds miscompilation bugs via a variant shader that renders a significantly different image compared to the image rendered by the associated reference shader. In this case `gsl-reduce` reverses as many of the transformations that were applied to the variant shader as possible while the difference persists. To create Vulkan CTS tests suitable for exposing such bugs we worked with the Amber developers to add AmberScript features related to comparing the outputs of multiple graphics pipelines. In particular, we added the ability to compare framebuffers in a *fuzzy* manner. This allows us to turn a GraphicsFuzz reference and reduced-variant shader pair into a single AmberScript file that (a) creates and runs a separate pipeline for each shader, rendering to distinct framebuffers, and (b) asserts fuzzy equality between these framebuffers.

A challenge associated with this is the selection of a suitable fuzzy comparison metric for our purpose. We collected a corpus of image pairs that – based on our shader compiler fuzzing experience – we would like to be deemed similar, and a set of pairs that we would like to be deemed different. The corpus includes image pairs produced by graphics drivers during our testing efforts, plus a few manually crafted image pairs that we believe could occur in theory and that we thought may prove challenging for certain comparison algorithms. We experimented with various image comparison algorithms provided by the `scikit-image` [47] Python library, including MSE, NRMSE, SSIM, and PSNR. We also tried several custom image comparison algorithms based on obtaining and comparing image histograms. We found that image histogram comparison was very effective at correctly classifying image pairs in our corpus, except for some manually crafted image pairs where one image was a rotation or mirror of the other. Indeed, the key weakness of image histogram comparison is that all spatial information is lost. A key advantage is it is very resilient to minor differences that other algorithms flag as important, but which we would typically like to be ignored. We chose to initially proceed with an image histogram comparison algorithm for the following reasons: it correctly classifies image pairs in our corpus as well as or better than most other algorithms; it is very simple to understand and implement (which is important because we don't want GPU vendors to struggle to understand why a Vulkan CTS test has failed and have to debug the image comparison algorithm itself); it has fairly low performance

requirements;⁹ with a high tolerance value, it is fairly forgiving of minor differences, and – to achieve a low false alarm rate – we would prefer to incorrectly classify an image pair as similar than incorrectly classify the pair as different (most image differences we encounter in practice are easily detected with a forgiving algorithm/tolerance).

We implemented our image comparison algorithm, *Histogram EMD* (where EMD stands for Earth Mover’s Distance [29]), in the Amber code base, and added a command to AmberScript of the form:

```
EXPECT buffer_1 EQ_HISTOGRAM_EMD_BUFFER buffer_2 TOLERANCE value
```

where `buffer_1` and `buffer_2` are framebuffers containing the images we wish to compare and `value` is the tolerance value. The test fails if the difference value returned by the algorithm exceeds the tolerance value.

These extensions to Amber provide a pathway for landing tests that expose wrong image bugs in CTS, and we have implemented the necessary scripts to directly generate such tests. We recently put several such tests up for Vulkan CTS code review, and a reviewer quickly found that the validity of one of the tests was questionable due to floating-point precision issues. We discuss this as Example 3 in §3.5. To err on the side of caution, we retracted the other wrong image tests we had put forward and manually simplified each one to double-check that it really did correspond to a driver bug rather than a floating-point precision issue. After sufficient manual simplification, we were able to add an Amber test for each of these bugs, consisting of a *single* shader (and a single pipeline) with a straightforward assertion to check that the single output image is red.

In order to be able to add wrong image tests with a pair of shaders to CTS with confidence, we are working on a corpus of reference shaders that are highly numerically stable.

3.5 Avoiding Invalid Tests

We are anxious not to waste Vulkan CTS reviewer time by proposing tests that turn out to be invalid and get rejected, or – worse – that get accepted (due to the invalidity being subtle, and not leading to failures on current drivers) and subsequently found to be invalid (necessitating their removal from every CTS release they have made it into). We discuss our main concerns related to possible invalid tests.

Preserving semantics during generation and reduction. As explained in §2.4, `GraphicsFuzz` produces a variant shader by having `gsl-fuzz` repeatedly apply semantics-preserving transformations to a reference, and upon finding a potential wrong image bug, invokes `gsl-reduce` to reduce the test case by repeatedly attempting to reverse or simplify transformations. For wrong image bugs, it is critical that all transformations preserve semantics both when applied and reversed/simplified. The way `GraphicsFuzz` has been designed, all information about the transformations that have been applied is recorded by `gsl-fuzz` via syntactic markers in the generated shaders. Examples of syntactic markers include using special preprocessor macros, and giving variables and functions special names or name prefixes. The `gsl-reduce` tool then needs to understand these markers and use them to reverse and simplify certain transformations without spoiling the syntactic markers that represent other transformations.

⁹ When running the Vulkan CTS on Android, the image comparison is done on the Android device using the CPU, which has some overhead, especially when using a simulated (software) CPU, as is commonly done when testing next-generation hardware.

In practice we have encountered several hard-to-diagnose bugs where `gsls-reduce` has erroneously changed the semantics of a shader, usually due to reversal of one transformation having messed up the syntactic markers associated with another transformation, which as a result gets incorrectly reversed.¹⁰

We guard against this in practice via a degree of manual inspection of the final reduced shader emitted by `gsls-reduce`, and as `gsls-fuzz` and `gsls-reduce` continue to become more stable this issue becomes less relevant. However, based on our experience, we regard having a separate generator and reducer that must understand one another in an intricate manner to be a serious pitfall of the `GraphicsFuzz` approach. Recent research on *internal* test case reduction has the potential to avoid the need for a separate generator and reducer [34], and could thus be useful in our domain.

Loop limiters. Recall that the *live code injection* transformation performed by `gsls-fuzz` (see §2.4) injects code from a donor shader into the shader under transformation in a manner such that the injected code really gets executed at runtime. A problem here is that the injected code may contain loops, and these loops may run for potentially large numbers of iterations. In particular, if the declarations of variables that control loop execution are not themselves injected, `gsls-fuzz` creates declarations for such variables and initializes them to randomized expressions, which can lead to infinite loops. Programs that risk containing infinite loops are used for compiler testing by tools such as `Csmith` [50], with the philosophy that it is better to accept that some programs will not terminate, and to use a timeout to bound the runtime of any individual test, than to put in place draconian measures to ensure that all loops terminate. Unfortunately, in the world of GPU shader compilers, long-running shaders cause display freezes, leading to the operating system’s GPU watchdog killing the executing shader. This can lead to the shader rendering what appears to be an incorrect image when in fact the image was simply incomplete.

We found that this problem confounded our test results, requiring significant manual inspection of final shaders to check for long-running loops. To overcome this we decided to go ahead and put a relatively draconian measure in place: every loop in every live-injected shader is truncated via a *loop limiter*. This is an additional counter variable specific to a loop. It is initialized to zero immediately before the loop. A conditional statement at the start of the loop body breaks from the loop if the counter exceeds a small positive value (randomly chosen at generation time), and increments the counter otherwise.

With reference to our discussion above about keeping the generator and reducer in sync: loop limiters are given special names when inserted by `gsls-fuzz`, and when simplifying live-injected code `gsls-reduce` checks for these names and takes care not to remove loop limiters unless removing the entire associated loop. Again, this coupling between generator and reducer is fragile and can be hard to maintain.

When reducing a compiler crash bug `gsls-reduce` aggressively shrinks a shader. In this case we allow it to remove loop limiters, which can mean that finally-reduced shaders may contain infinite loops. While the resulting shaders are good enough to reproduce a compiler crash, they are not suitable for addition to CTS, as all CTS tests should be runnable. We therefore inspect shaders manually and edit them to avoid any infinite loops – while preserving the compiler crash – before submitting them for CTS review.

¹⁰ See <https://github.com/google/graphicsfuzz/pull/599> as an example pull request that fixes such an issue.

Array bounds clamping. Live-injected code may also contain access into arrays and vector/matrix types, which have the potential to be out-of-bounds if their indexing expressions depend on variables that `gls-fuzz` initializes to randomized expressions. SPIR-V for Vulkan requires that all accesses are in-bounds. Fortunately, array and vector/matrix sizes are always known statically in GLSL and there are no pointers in the language. We therefore rewrite every array index expression e that appears in live-injected code as `clamp(e, 0, N - 1)`, where N is the size of the array or vector/matrix being accessed, and `clamp(a, b, c)` is the GLSL built-in that clamps a into the range $[b, c]$. An exception to this is when e is a literal that is already in-bounds. As with loop limiters, `gls-reduce` is responsible for preserving these in-bounds clamping expressions during test case reduction.

Floating-point stability. We use an example to illustrate the risk of submitting invalid CTS tests posed by floating-point instability.

► **Example 3.** A transformation that `GraphicsFuzz` may try to apply is to replace a floating-point expression e with an expression e/ONE , where ONE is an expression guaranteed to evaluate to 1.0 at runtime. `GraphicsFuzz` has many possible ways of synthesizing an expression that is expected to evaluate to 1.0, one method being to generate an expression of the form `length(normalize(v))`, where v is some non-zero vector. The `normalize` GLSL built-in yields a unit vector (when applied to a non-zero vector), and `length` yields the length of a vector, so the expression intuitively should evaluate to 1.0. However, it turns out that the floating-point precision requirements on SPIR-V instructions mean that the result might not *quite* evaluate to 1.0; some round-off error is allowed [20, pp. 1754–1759].

We thought we had found a wrong image bug in `SwiftShader` upon finding a major image difference to be caused by transforming the following code snippet:

```

1 // 'ref' and 's' are 'float' variables; 'ref' has value 32.0 at runtime
2 for (int i = 1; i < 800; i++) {
3     // 'mod' is the floating-point modulus operation
4     if (mod(float(i), ref) <= 0.01) {
5         s += 0.2;
6     }
7     ...
8 }
```

This code snippet causes `s` to increase by 0.2 every time `i` is a multiple of 32, since this is the only scenario where `mod(float(i), ref)` will be sufficiently small for the `if` condition to evaluate to true. `GraphicsFuzz` replaced `ref` with `ref / length(normalize(vec3(...)))`, where the `...` is a placeholder for a non-trivial but sensible expression that evaluates to 1.0 (so that the resulting vector is (1.0, 1.0, 1.0)).

What we assumed was a bug in `SwiftShader` turned out to be a false alarm. After some manual analysis we found that the divisor `length(normalize(vec3(...)))` evaluated to a value *slightly larger* than 1.0, so that the second argument to the floating-point `mod` built-in was *slightly smaller* than 32.0 (due to `ref` being exactly 32.0). As a result, the statement `s += 0.2` became *unreachable*, even for loop iterations where `i` is a multiple of 32 since the modulus of a multiple of 32 with a value v slightly smaller than 32.0 leads to the value v .

Floating-point precision issues like this hammer home the importance of using numerically stable shaders when searching for wrong image bugs using `GraphicsFuzz` – the code snippet in Example 3 demonstrates that the shader in question was *not* numerically stable. It is also important to maximize the extent to which the transformations that `GraphicsFuzz` applies actually preserve floating-point semantics. The deliberately ambiguous approach that graphics shading languages take to floating-point (in order to accommodate many disparate

GPUs) means that we can never be certain that a program transformation will completely preserve semantics (since it can affect the optimizations the shader compiler performs, and those optimizations are permitted to have small effects on floating-point results). However, where possible we try to take measures to avoid floating-point error; for instance we have changed the representation of 1.0 discussed in Example 3 from `length(normalize(v))` to `round(length(normalize(v)))`, where the `round` GLSL built-in rounds its floating-point argument to the nearest integer value; this ensures that the result will indeed be 1.0.

4 **gfauto**

`gfauto` (short for GraphicsFuzz auto) is a set of tools for using GraphicsFuzz in a “push-button” fashion with minimal interaction, geared towards generation of tests that can be added to CTS using the pathways described in §3. Pre-`gfauto`, performing a fuzzing run required manually generating a set of variant shaders offline from a set of reference shaders, followed by a number of manual steps to run the reference and variant shaders on target devices, waiting for the shaders to finish, and then manually triggering reductions of interesting variant shaders. This approach is unnecessarily inefficient when the main objective is to find as many interesting variants (i.e. those that expose bugs) for a given device as possible within a fuzzing run. In contrast, the high-level, *automatic* workflow of `gfauto` is: generate a variant shader from a reference shader; run the shaders on the target device; reduce the variant if it is interesting, otherwise discard it; repeat. This process can run continuously for long periods of time, without interaction, which maximizes the number of interesting variant shaders, and thus the potential number of new CTS tests. Using `gfauto` greatly decreases the length of time needed to perform a fuzzing run and submit a number of CTS tests from that run; we estimate the time period has gone from about 1-2 days (pre-`gfauto`) down to 1-2 hours (when using `gfauto`).

We detail three key features of `gfauto`: creation and replay of self-contained tests (§4.1), bug de-duplication and prioritization (§4.2), and automatic Vulkan CTS test export (§4.3).

4.1 Creation and replay of self-contained tests

Pre-`gfauto`, the output of a fuzzing run was a directory of images and log files from running reference and variant shaders; the reference and variant shaders themselves were stored in a different directory. Collecting the shaders and output files needed to reproduce and investigate a bug required copying files from different directories, and these files were stored in an ad-hoc format. Furthermore, the versions of the tools required in order to run the test (such as `glslang` and `spirv-opt`) were not captured. Details about the target device were available but were again outputted in yet another directory and were typically archived in an ad-hoc format, if at all. Thus, reproducing and investigating a bug was difficult and time-consuming, and useful information was often lost.

`gfauto` generates a self-contained test from the start. The generated test directory contains a `test.json` metadata file and the reference and variant GLSL shaders. The metadata file contains all information needed to run the test, including a list of required tools and their versions (which are downloaded on-the-fly), an error signature for the test (described below, and initially empty until a crash or wrong image is observed), details of the device on which the test should be run (including the driver version), and the steps needed to run the test (e.g. running `spirv-opt` with a given series of optimization passes). In particular, `gfauto` runs the test for the first time using the test metadata file, and is restricted to the tools specified in the metadata; this ensures that no tool dependency can be missed. A test directory can

thus be replayed with a single command. In the case of Android, the `test.json` file even captures the Android device serial number so that the test can be automatically replayed on the target device, with no interaction, as long as it is connected to the host machine.

4.2 Bug de-duplication and prioritization

A fuzzing run pre-`gfauto` would often find a large number of crash-inducing variant shaders, but upon inspection of crash stack traces it would become apparent that many variants were exposing the same bug. Clearly, we would like to prioritize the *unique* bugs found. We wrote several ad-hoc scripts to classify variants that caused crashes into unique “buckets”, so that each bucket represents a unique bug (based on the top function name in the stack trace). However, this process was still tedious (as it involved several manual steps) and unreliable (as the scripts were typically hand-tuned for a given fuzzing run). Furthermore, this classification was never made permanent, so the information would typically be absent in future fuzzing runs. Thus, we would often re-find bugs that had already been found in previous runs and we would have to manually avoid investigating these.

In `gfauto`, generated tests that expose bugs are stored in buckets in the file system, where a bucket is a directory named using the “signature” (usually the top function name in the stack trace). This makes it trivial to identify tests that expose unique bugs (pick one test from each bucket). The signature is also stored in the test metadata, ensuring the information is never lost, even if the test is moved. A Python function `get_signature` takes the log contents as its only input and outputs the signature string; we update this function as needed to get an accurate bug signature in a number of scenarios. For example, if a stack trace is present (in one of several different formats), the top function name of the stack trace is used, if available, falling back to the hex offset of the function otherwise. Alternatively, if a recognized error message or assertion failure pattern is seen, the error message itself can be used as the signature. This approach ensures we reliably classify tests in most cases. A configurable threshold ensures only a small number of tests are stored in each bucket; subsequent tests are discarded and, crucially, do not need to be reduced, which is expensive. Additionally, `gfauto` supports downloading and running our Vulkan CTS tests on the target device, capturing the signatures (if an error occurs), and ignoring these signatures during the next fuzzing run. This allows `gfauto` to ignore bugs that can already be found by existing tests, even if the signatures change between fuzzing runs; this might happen due to a graphics driver update on the target device or due to changes in `gfauto`’s `get_signature` function. In particular this allows unfixed bugs found in previous fuzzing runs to be ignored, assuming appropriate CTS tests were created.

Bug de-duplication challenges. The above approach works well most of the time, but some issues remain. Some bugs are nondeterministic in nature. In particular, some of our tests appear to trigger memory leaks in certain shader compilers, which can cause an abort to occur at arbitrary places. Our `gsl-reduce` tool runs the test up to five times initially (before commencing reduction) in order to validate that the the originally-observed crash signature can be reproduced. Highly nondeterministic tests will often fail this validation step, as the signature will be different every time.

Another issue is when a driver returns a “shader compile error” or “shader link error” message, even though the provided shaders are valid. The driver often provides no additional information, and so there is no way to further distinguish the shader compiler bug. Thus, if we find hundreds of “shader compile error” bugs, we may have found hundreds of distinct compiler bugs, or just one, or any number in between. The same issue applies for tests that expose wrong image bugs, which are simply given a signature of “`wrong_image`”. In future

work, we hope to identify tests that likely expose distinct wrong image bugs by comparing the semantics-preserving transformations that remain in the fully-reduced variant shaders. Tests that contain very distinct transformations are perhaps more likely to be triggering different shader compiler bugs than tests that contain similar transformations. For compile/link errors (where reduction need not be semantics-preserving) we may be able to use a similarity measure on fully-reduced shaders for de-duplication purposes, drawing on ideas for “taming” compiler fuzzers [10].

4.3 Vulkan CTS test export

Creating a Vulkan CTS test from a bug found by GraphicsFuzz (using `gfauto` or otherwise) requires some manual iteration on the test. As explained in §4.1, reproducing and investigating a bug pre-`gfauto` was time-consuming, and information was liable to be lost. The self-contained nature of a `gfauto` test greatly improves the experience. Iterating on a CTS test typically requires tweaking the original GLSL shaders and re-generating the SPIR-V (using the correct versions of `glslang` and `spirv-opt`) again and again. As already stated above, we believe this has greatly decreased the time needed to get from a fuzzing run to a number of submitted CTS tests, from about 1-2 days (pre-`gfauto`) down to 1-2 hours (when using `gfauto`).

We took the push-button nature of `gfauto` further by automating the end-to-end process of adding a Vulkan CTS test (after manually tweaking the GLSL shaders). Alongside each `gfauto` test, we store a Python script that generates the final `.amber` file for the Vulkan CTS test. The `.amber` file is similar to the one generated when running the test, but with a copyright header and, as illustrated in Figure 6, a short description and a comment explaining *why* the test passes; note that the short description and comment are manually written by us. The Python script includes the name of the output `.amber` file, the contents of these comments, and some optional tweaks, such as additional AmberScript commands that we might want to add to provide an oracle for the test. Another utility tool then takes this `.amber` file and inserts it into the Vulkan CTS source tree, taking care of updating various index files based on the `.amber` file name and the short description comment. This yields a patch that can be directly put up for Vulkan CTS code review.

5 Finding Test Coverage Gaps Using GraphicsFuzz and `gfauto`

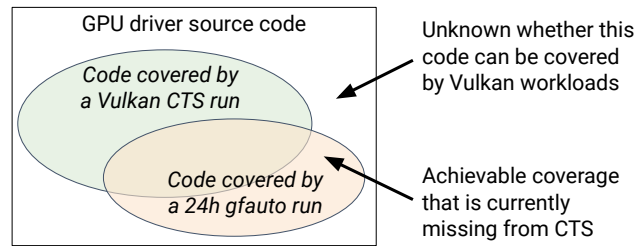
5.1 Absolute Code Coverage and its Limitations

Line coverage is a widely-used metric for assessing the adequacy of a test suite at a basic level. While many more thorough notions of coverage have been proposed [1], line coverage is appealing because it is both simple to compute and *actionable* [4] – a lack of line coverage can typically be addressed by crafting appropriate tests. A simple idea for growing Vulkan CTS is therefore to run CTS on an open source Vulkan driver and then attempt to write tests to cover parts of the driver that are not reached.

This simple idea suffers from two key problems:

1. It might be *inappropriate* for a Vulkan CTS test to reach certain driver code;
2. For code that could be covered in principle, it is likely very labour-intensive to manually write tests that cover it in practice.

To illustrate problem 1, a recent run of Vulkan CTS on the open source Mesa driver with an AMD back-end [11] identified much uncovered code, but a lot of this code turned out to be (a) debug code (such as routines for dumping data structures in text format), (b) code specific to APIs other than Vulkan (such as OpenCL), and (c) code specific to non-AMD GPUs. It is perfectly legitimate for this code to remain uncovered.



■ **Figure 8** Illustration of differential code coverage. Driver code covered during a `gfauto` run but not during a CTS run is code that *can* be exercised but for which CTS test coverage is lacking. The tests that `gfauto` generated to achieve such coverage provide a basis for new CTS tests.

5.2 Differential Code Coverage

To partially solve problem 1 of §5.1 we appeal to *differential* code coverage. Suppose we know which lines of an open source Vulkan driver are covered during a CTS run; call these lines A . Suppose further that we know which lines of the driver are covered by running some other valid Vulkan workload, such as a Vulkan-based game, or a 24-hour run of `gfauto`; call these lines B . For any line $l \in B \setminus A$, we know that l *can* be exercised by valid use of the Vulkan API, so the fact that a CTS run does not exercise l demonstrates a coverage gap in CTS that can certainly be plugged in principle.¹¹

The idea of using `gfauto` and differential coverage to identify code that CTS could in principle cover is illustrated in Figure 8. One might also imagine that differential coverage analysis could be used to drive improvements in `GraphicsFuzz`: code that CTS can cover but that `gfauto` cannot might indicate that `gfauto` should be seeded with a richer set of reference shaders, or that `GraphicsFuzz` should implement more adventurous transformations. However, the scope of `GraphicsFuzz` is limited to *shader compilers*, while CTS tests the whole of the Vulkan API, so some knowledge of which parts of the driver relate to shader compilation specifically would be required.

Although the idea of differential coverage analysis is not new (e.g., continuous integration systems often provide facilities for visualizing the coverage trajectory of a project), we could not find a suitable open source project that provides it, so we implemented our own tooling for differential coverage, which we describe in §5.5.

5.3 Using Test Case Reduction to Synthesize Small Tests

While differential coverage helps with problem 1 of §5.1, it does not help directly with problem 2: just knowing that a line is coverable in principle does not yield a suitable CTS test that covers the line. If workload B (see §5.2) were an interactive game, it might be very difficult to reverse-engineer a stand-alone test that provides coverage of a particular line.

However, if workload B is a `gfauto` run, we can at least obtain a number of `GraphicsFuzz`-generated variant shaders that provide new coverage. Adding these tests to CTS would serve to fill the coverage gap, but recall from §3.2 that generated tests are very large, and virtually impossible for humans to understand in practice, thus unsuitable for direct addition to CTS.

To overcome this problem, we appeal to *test case reduction* in the following manner. Having performed a `gfauto` run for, say, 24 hours, and identified a set of driver source code lines $B \setminus A$ that were reached by `gfauto` but not by CTS, we manually choose one such line

¹¹ It is theoretically possible that, e.g. due to concurrency, reachability of line l might be nondeterministic, but we have not encountered this in practice.

and prefix it with `assert(false)` – i.e., we pretend that it is *erroneous* to reach the line. We recompile the driver without coverage instrumentation and run `gfauto` again using the same parameters (random seed and corpus of shaders) as in the original run. `gfauto` will, once again, reach the line, this time leading to an assertion failure. `gfauto` will treat the assertion failure as a shader compiler crash and invoke `gsl-reduce` to shrink the shader to a minimal form that still covers the line. The minimized shader is an excellent candidate for being added to CTS since it is small enough to be human-readable. The process is repeated by choosing another line from $B \setminus A$, avoiding lines that we believe are likely to already be covered by the candidate CTS tests found so far. We periodically re-run CTS after adding the new tests to update workload A , thus ensuring we don't miss any coverage gaps.

At present we have been adopting this approach by collecting differential line coverage of `SwiftShader`, which incorporates `spirv-opt` and large parts of LLVM internally. The fact that `SwiftShader` is open source simplifies the process considerably, although it should be possible to apply a similar process to closed-source binary drivers using instruction coverage instead of line coverage, and by overwriting an instruction with an interrupt instruction (or some invalid instruction) instead of prefixing a line with `assert(false)`.

Custom interestingness test. Like many reducers, `gsl-reduce` supports a custom “interestingness test” script that signals to the reducer whether the shader that is being reduced is still “interesting” (e.g. still crashes the driver). Thus, instead of modifying the driver source code, we could simply provide an interestingness test that runs the shader using the driver with coverage instrumentation, processes the coverage data, and checks if the line of interest was covered. Unfortunately, processing the coverage data is slow, and thus usually done offline. `gsl-reduce` will typically run the interestingness test hundreds or thousands of times before finding a minimal shader. Thus, making the driver crash via an assertion failure is a much faster approach and, conveniently, already triggers a reduction in `gfauto` without requiring a customized interestingness test.

Less coverage after reduction. A potential downside of our approach is that, after reduction, a shader may cover fewer lines of interest than before. For example, an unreduced shader might cover three seemingly unrelated functions, `f`, `g`, and `h`, that are all not covered by CTS, while the reduced shader might cover just one of the functions, `f`, because we only added an assertion in `f` and thus the reducer did not try to preserve coverage of `g` and `h`. Although this may seem undesirable, focusing on just one function at a time typically allows the reducer to go further (potentially *much* further) in minimizing the shader. We would much rather have three *simple* and *small* CTS tests, each covering a different function, than one *complex* and *large* CTS test that covers all three functions.

5.4 Manually Tweaking Tests to Improve Oracles

Although the reduced tests could be added to the CTS directly, this is almost never appropriate. As with crash bugs, we need to add an oracle to the test, else a driver that does nothing could pass the test. Again as with crash bugs, we typically add code to the shader to make it render the colour red and add a check to the test to ensure all rendered pixels are indeed red. However, the test can be made much more useful if the newly covered lines affect the output colour value so that if a bug was introduced in the newly covered lines, the test would fail. Also note that a coverage gap test will fill a coverage gap for the Vulkan driver that we were running (e.g. `SwiftShader`), but the hope is that it may also be a meaningful test for other drivers, especially if it relates to a feature for which test coverage is generally lacking. The test should be written with this in mind, as we will see below.

22:22 Putting Randomized Compiler Testing into Production

► **Example 4.** The following is a generated, reduced fragment shader that covers constant folding code in `SwiftShader` that replaces a dot product call with zero if one of the operands is a vector of zeros:

```
1 void main() {
2     if(1.0 >= dot(vec2(1.0, 0.0), vec2(0.0, 0.0))) {
3         _GLF_color = vec4(1.0);
4     }
5 }
```

To transform the shader into a form suitable for the Vulkan CTS, we could simply add code to the end of `main` that assigns the colour red to `_GLF_color`, but this has two key disadvantages: (1) any driver that *incorrectly* constant folds the dot function call will still always pass the test, and; (2) a driver might eliminate everything above our final write to `_GLF_color`, and thus, on this hypothetical driver, the test would not even cover code related to the dot product operation. A simple fix is to use the output of the dot function call in the output colour value, as follows:

```
1 void main() {
2     float zero = dot(vec2(1.0, 0.0), vec2(0.0));
3     _GLF_color = vec4(1.0, zero, 0.0, 1.0); // we expect red
4 }
```

However, even this is not ideal; if a driver incorrectly replaced the dot call with a negative float value, the output colour would still be red, as the output colour components are clamped by the driver to be between 0 and 1, as required by the Vulkan specification. In our final version of the test,¹² we check that the result of the dot call is exactly 0, and we only output red in this case:

```
1 void main() {
2     if(dot(vec2(1.0, 0.0), vec2(0.0)) == 0.0) // precise check
3         _GLF_color = vec4(1.0, 0.0, 0.0, 1.0); // we expect red
4     else
5         _GLF_color = vec4(0.0);
6 }
```

The process of manually changing the test to be useful is non-trivial and probably cannot be automated as it requires some creativity, but the reduced test synthesized by `gfauto` is an excellent starting point and is usually not that different to the final version of the test.

5.5 Implementing Differential Code Coverage

The GCC compiler supports compiling an application with *coverage instrumentation*, causing coverage data to be output when the application runs, which can then be processed with the `gcov` tool. For example, to capture line coverage of `SwiftShader` when running the Vulkan CTS, we could perform the following steps:

- **Compile `SwiftShader` with GCC, adding the `--coverage` flag.** This builds the `SwiftShader` Vulkan library with coverage instrumentation. For each `.o` file that was written, the compiler also writes a `.gcno` file at the same location. The `.gcno` (`gcov` notes) files describe the control flow graph of the corresponding `.o` files, and include mappings to source code file paths and line numbers.

¹²<https://github.com/KhronosGroup/VK-GL-CTS/blob/master/external/vulkancts/data/vulkan/amber/graphicsfuzz/cov-const-folding-dot-determinant.amber>

- **Run the Vulkan CTS using SwiftShader.** Due to the coverage instrumentation in the SwiftShader library, `.gcda` files are output alongside the corresponding `.o` and `.gcno` files. The `.gcda` (gcov data) files contain the control flow graph block and edge execution counts (i.e. the number of times each block and edge was executed).
- **Run gcov to process the .gcno and .gcda files to get line coverage information.** Manually executing gcov on every `.gcno` file from the required directory (or directories) while avoiding output filename clashes is a tedious process. Additionally, the line coverage output from gcov is fairly primitive. Thus, there are third-party tools, such as `lcov` and `gcovr`, that invoke gcov automatically, yielding information in an intermediate data format, and then further process this data to generate, say, an HTML report that shows every source file annotated with the execution count of each line.

Unfortunately, we could not find any tools capable of obtaining *differential* line coverage as described in §5.2. Thus, we created our own set of tools¹³ that are similar in spirit to `lcov` and `gcovr`, but support obtaining differential line coverage.

`cov_from_gcov` processes `.gcno` and `.gcda` files into a single `.cov` output file that contains the set of source file lines that were executed. The `.gcno` and `.gcda` files are processed by invoking gcov in each required directory to output intermediate data files that are then processed further to produce the `.cov` output file.

`cov_new` takes `A.cov` and `B.cov` as inputs, and outputs the differential coverage to `new.cov`. The `new.cov` file is simply `A.cov` minus `B.cov` (i.e. $A \setminus B$ from §5.2).

`cov_to_source` takes `new.cov` as its input, and outputs two parallel directory structures `zero/` and `new/`. Both directories contain copies of the original source files with a prefix added to every line: the `zero/` directory has a “0” prefix added to every line; the `new/` directory has a “1” prefix added to every line present in `new.cov`, and a “0” prefix for every other line. The two directory structures can be compared using a diff tool; lines that are different are the lines of interest (i.e. the lines in $A \setminus B$ from §5.2), and will be highlighted. The approach of generating parallel directory structures means we avoid generating large HTML reports, which can be slow to open and navigate, and instead allows the use of any existing diff tool that is already optimized for this type of task.

6 Fuzzing the SPIR-V Tooling Ecosystem

Recall the rich ecosystem of SPIR-V-related tools shown in Figure 2. Because `gfauto` uses many of these tools during a fuzzing run, we have the potential to find bugs in them as well as finding bugs in vendor shader compilers. Furthermore we have also conducted some fuzzing runs that include the `spirv-cross` tool (not part of the default `gfauto` workflow).

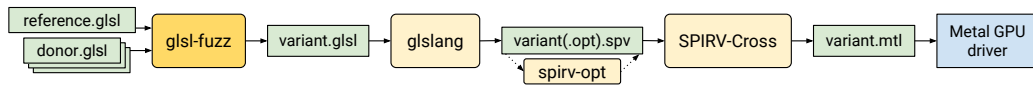
We illustrate, via examples, both the strength of being able to find bugs in multiple tools and the challenge associated with determining which tool is to blame when a problem arises.

► **Example 5.** Running `spirv-opt` on a shader generated by `gfauto` led to a non-zero exit code. We reported this bug to the SPIRV-Tools project,¹⁴ assuming it to be a bug in `spirv-opt`. The `spirv-opt` authors investigated and determined that in fact the shader contained *invalid* SPIR-V that `spirv-val` (which `gfauto` runs at every transformation stage) had missed. This identified a validator bug (in the form of a validator omission) but raised the question of what had created the invalid SPIR-V. It turned out that `gfauto` had run `spirv-opt` as part of

¹³<https://github.com/google/graphicsfuzz/blob/master/gfauto/docs/coverage.md>

¹⁴<https://github.com/KhronosGroup/SPIRV-Tools/issues/3031>

22:24 Putting Randomized Compiler Testing into Production



■ **Figure 9** Fuzzing Metal drivers from GLSL.

generating the shader, and its block merging pass had been too aggressive: loops in SPIR-V assembly have designated *merge* and *continue* blocks, the *merge* block denoting the loop’s exit, and the *continue* block denoting the start of a region of code that must be traversed in order to return to the loop head. The block merging pass was allowing the merge block of one loop and the continue block of another loop to be merged. This turned out to be illegal according to the SPIR-V specification, though the wording of the specification did not spell the rule out very clearly. The SPIRV-Tools team enhanced `spirv-val` to detect this kind of invalidity, and fixed the bug in `spirv-opt`’s block merging pass.¹⁵

This resolved the combined validator and optimizer bug that we had found with `gfauto`. Unfortunately, it turned out that 55 existing Vulkan CTS tests contained SPIR-V that was invalid for the same reason – in many cases the SPIR-V in question had been processed by `spirv-opt`’s previously buggy block merging pass. This necessitated fixing these tests in the master branch of CTS, as well as in multiple release branches.

The SPIR-V working group are discussing how to clarify the specification with respect to its rules about the structure of loops and other control flow constructs, in part due to this (and other) reports from our fuzzing efforts.

► **Example 6.** An early assertion failure that we triggered in `spirv-opt`,¹⁶ by fuzzing using a random combination of optimizer flags, turned out to be due to a “merge return” optimization pass being applied to a SPIR-V control flow graph that it was known not to be able to handle. The SPIRV-Tools team hardened `spirv-opt` by having the “merge return” pass explicitly check for unsupported control flow graph idioms and, on encountering an unsupported idiom, gracefully exit with an error informing the user that they should run the “eliminate dead code” pass first.

Subsequently, we ensured that `gfauto` only generates lists of `spirv-opt` optimization passes in which “merge return” (if present) runs after “eliminate dead code”.

► **Example 7.** We ran some experiments testing MoltenVK [26], an implementation of most of Vulkan on top of Apple’s Metal graphics API, on a MacBook Pro. MoltenVK uses `spirv-cross` to translate SPIR-V to the Metal shading language (MetalSL) so it can be sent to the Metal shader compiler within the Metal driver on a Mac or iOS device. The full translation pipeline is illustrated in Figure 9. When a wrong image is produced in this setup, the bug could be in the Metal driver or in any of the tools that come before (shown as rounded-rectangles in Figure 9). Differential testing can come to the rescue here: if the bad image is also produced by the variant shader in a “vanilla” setup, e.g. by using `glslang` and rendering the resulting SPIR-V using some other Vulkan driver, the bug is very likely in `glsl-fuzz` or `glslang`. Otherwise, if the bug manifests only when adding the same `spirv-opt` passes (before running on this other Vulkan driver), the problem is likely in `spirv-opt`. Otherwise, the bug is likely in `spirv-cross` or the Metal driver.

¹⁵ <https://github.com/KhronosGroup/SPIRV-Tools/pull/3068>

¹⁶ <https://github.com/KhronosGroup/SPIRV-Tools/issues/1962>

We found such a wrong image bug in our testing, and used differential testing to conclude that the bug was likely in `spirv-cross` or the Metal driver. After inspecting the MetalSL code, we found it to be incorrect and so ascertained that the bug was in `spirv-cross`. We submitted a bug report¹⁷ and the bug was promptly fixed.

7 Related Work

Randomized and metamorphic compiler testing techniques. Randomized testing of compilers has a long history; see e.g. [21] for a very early example, and multiple surveys [8, 3, 30]. Random differential testing (RDT) of C compilers was investigated to some extent by McKeeman [36], and the Csmith project from the University of Utah [50] has triggered a lot of interest in the topic over the last decade. An early approach to metamorphic compiler testing involved generating equivalent programs from scratch [46]. More recent work on *equivalence modulo inputs* testing (EMI) [31, 44], a form of metamorphic testing, showed that approaches based on transforming programs in a manner that preserves semantics at least for certain inputs can be an effective way of triggering wrong code bugs. The RDT and EMI approaches have been extended to allow testing of OpenCL compilers [33], and the EMI approach was the inspiration for the approach to metamorphic testing employed by `GraphicsFuzz` [16, 15], on which the work described in this experience report was built. Randomized differential testing has also been applied to other program processing tools, such as refactoring engines [14] and static analyzers [13], and there is scope for applying techniques from compiler testing to program analyzers more generally [6].

Experience reports related to compiler testing. A short report on work at the UK's National Physical Laboratory describes some experiences testing compilers for Pascal, Ada and Haskell using random program generators, mainly focusing on the relative difficulty of constructing program generators for each of these languages [48]. McKeeman's seminal paper on differential testing includes a section on randomized compiler testing that is written in the style of an experience report [36]. In comparison to our paper, these reports do not discuss the challenges of setting up a pathway from randomly-generated test cases to test cases suitable for incorporation into a standard compiler test suite. An edited volume on validation of Pascal compilers provides a number of experience reports related to the testing and validation process [49]. These reports discuss issues related to constructing compiler conformance tests, but do not mention randomized testing. Furthermore, none of the aforementioned experience reports discuss the challenges of testing *graphics* compilers.

Empirical studies related to compiler bugs and compiler testing. There have been three recent empirical studies related to compiler bugs and randomized compiler testing: a study on the relative effectiveness of compiler testing based on RDT vs. EMI testing [7], a study on the characteristics of bugs in the GCC and LLVM compilers (not specifically focusing on bugs found via randomized testing) [45], and a study that aims to assess the relative impact on end-user software of fuzzer-found compiler bugs compared with compiler bugs encountered and reported "in the wild" by users [35]. Unlike our work these studies all focus on C/C++ compilers, not compilers for graphics shading languages. A main finding from [35] – that bugs found by fuzzers appear to have at least as much practical impact as bugs reported by users – supports our belief that adding fuzzer-found compiler bugs to compiler regression test suites is a worthwhile endeavour.

¹⁷<https://github.com/KhronosGroup/SPIRV-Cross/issues/1091>

Compiler test case reduction and bug de-duplication. The *semantics-changing* reduction mode of `gsl-reduce` is similar in nature to the approach taken by the C-Reduce tool [40], following the well-known delta debugging method [51]. A difference between `gsl-reduce` and C-Reduce is that `gsl-reduce` exclusively uses valid abstract syntax tree transformations to reduce a shader, whereas C-Reduce uses a combination of methods, including language-aware transformations built on top of the Clang framework, language-agnostic transformations based on line and token deletion, and methods in-between that assume only basic language properties, such as that the language is block-structured, with blocks delimited by braces. As a result, C-Reduce can be applied to programs from a variety of languages (e.g. it has been successfully applied to OpenCL C [39]), while `gsl-reduce` is specific to GLSL. It would be interesting to investigate how well C-Reduce works for the reduction of GLSL programs that induce shader compiler crashes.

The semantics-preserving mode of `gsl-reduce` is intimately tied to the semantics-preserving transformations applied during metamorphic testing. As discussed in §3.5, the tight coupling between `gsl-fuzz` and `gsl-reduce` related to this mode has made it hard to maintain the pair of tools. Recent work proposes leveraging a test-case generator to provide test case reduction “for free”, by repeatedly re-generation to search for smaller tests that still trigger a bug [34]. An approach along these lines may be effective in avoiding the need for a tightly coupled generator and reducer in our domain.

Work on automated ranking of compiler bug reports proposes several metrics that can be used to order bug-inducing tests, with the aim of presenting a diverse selection of test cases exposing distinct bugs first [10]. Our de-duplication of crash bugs based on crash signatures (see §4.2) has not yet required this level of sophistication, but we believe such techniques could be brought to bear for de-duplication of wrong image bugs for which there is no analogue to a crash signature.

8 Conclusions and Future Work

We have described our experience rolling out graphics shader compiler fuzzing, based on the `GraphicsFuzz` tool chain, in a production environment with the goal of improving the Vulkan Conformance Test Suite via new tests that expose shader compiler bugs or provide additional coverage of shader processing tools. We hope the various insights in this report will be useful to researchers interested in testing programming language implementations.

We identify several directions for future practical work in this area.

Direct fuzzing for SPIR-V. We have gotten significant mileage from testing SPIR-V shader compilers via GLSL shaders, but the SPIR-V features this flow will exercise are inevitably limited, motivating the need for a fuzzer that works at the SPIR-V level.

Stability tests. We discuss the avoidance of invalid tests in §3.5. However, unintentionally invalid tests (due to bugs in the `GraphicsFuzz` tooling) have sometimes led to the discovery of serious driver stability issues, e.g. Android devices rebooting after accessing invalid memory, or failing to gracefully recover from long-running shaders [15]. It would be valuable to put in place a suite of tests, distinct from Vulkan CTS, to check that invalid shaders cannot derail an operating system.

Higher confidence in wrong image bugs. As discussed in §3.4 we are presently exercising caution regarding adding wrong image tests to CTS. A corpus of highly numerically stable shaders would allow us to proceed with greater confidence here, as would a more detailed analysis of the possible floating-point effects of the transformations that `gsl-fuzz` employs.

References

- 1 Paul Ammann and Jeff Offutt. *Introduction to Software Testing*. Cambridge University Press, 2 edition, 2017.
- 2 Apple. About the security content of ios 10.3, 2017. see “Processing maliciously crafted web content may result in the disclosure of process memory”. URL: <https://support.apple.com/en-gb/HT207617>.
- 3 Abdulzееz S. Boujarwah and Kassem Saleh. Compiler test case generation methods: a survey and assessment. *Information & Software Technology*, 39(9):617–625, 1997. doi:10.1016/S0950-5849(97)00017-7.
- 4 Arkady Bron, Eitan Farchi, Yonit Magid, Yarden Nir, and Shmuel Ur. Applications of synchronization coverage. In Keshav Pingali, Katherine A. Yelick, and Andrew S. Grimshaw, editors, *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP 2005, June 15-17, 2005, Chicago, IL, USA*, pages 206–212. ACM, 2005. doi:10.1145/1065944.1065972.
- 5 bugs.chromium.org. Issue 675658: Security: Malicious WebGL page can capture and upload contents of other tabs, 2016. URL: <https://bugs.chromium.org/p/chromium/issues/detail?id=675658>.
- 6 Cristian Cadar and Alastair F. Donaldson. Analysing the program analyser. In Laura K. Dillon, Willem Visser, and Laurie Williams, editors, *Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, May 14-22, 2016 - Companion Volume*, pages 765–768. ACM, 2016. doi:10.1145/2889160.2889206.
- 7 Junjie Chen, Wenxiang Hu, Dan Hao, Yingfei Xiong, Hongyu Zhang, Lu Zhang, and Bing Xie. An empirical comparison of compiler testing techniques. In Laura K. Dillon, Willem Visser, and Laurie Williams, editors, *Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, May 14-22, 2016*, pages 180–190. ACM, 2016. doi:10.1145/2884781.2884878.
- 8 Junjie Chen, Jibesh Patra, Michael Pradel, Yingfei Xiong, Hongyu Zhang, Dan Hao, and Lu Zhang. A survey of compiler testing techniques. *ACM Computing Surveys*, 2020. To appear.
- 9 T.Y. Chen, S.C. Cheung, and S.M. Yiu. Metamorphic testing: a new approach for generating next test cases. Technical Report HKUST-CS98-01, Department of Computer Science, The Hong Kong University of Science and Technology, 1998.
- 10 Yang Chen, Alex Groce, Chaoqiang Zhang, Weng-Keen Wong, Xiaoli Z. Fern, Eric Eide, and John Regehr. Taming compiler fuzzers. 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 197–208. ACM, 2013. doi:10.1145/2491956.2462173.
- 11 Igalia / codecov.io. Coverage report for vulkan cts on open source mesa driver with amd bck-end, 2020. URL: <https://codecov.io/gh/Igalia/mesa/>.
- 12 Keith Cooper and Linda Torczon. *Engineering a Compiler*. Morgan Kaufmann, 2002.
- 13 Pascal Cuoq, Benjamin Monate, Anne Pacalet, Virgile Prevosto, John Regehr, Boris Yakobowski, and Xuejun Yang. Testing static analyzers with randomly generated programs. In Alwyn Goodloe and Suzette Person, editors, *NASA Formal Methods - 4th International Symposium, NFM 2012, Norfolk, VA, USA, April 3-5, 2012. Proceedings*, volume 7226 of *Lecture Notes in Computer Science*, pages 120–125. Springer, 2012. doi:10.1007/978-3-642-28891-3_12.
- 14 Brett Daniel, Danny Dig, Kely Garcia, and Darko Marinov. Automated testing of refactoring engines. In Ivica Crnkovic and Antonia Bertolino, editors, *Proceedings of the 6th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2007, Dubrovnik, Croatia, September 3-7, 2007*, pages 185–194. ACM, 2007. doi:10.1145/1287624.1287651.
- 15 Alastair F. Donaldson, Hugues Evrard, Andrei Lascu, and Paul Thomson. Automated testing of graphics shader compilers. *PACMPL*, 1(OOPSLA):93:1–93:29, 2017. doi:10.1145/3133917.

- 16 Alastair F. Donaldson and Andrei Lascu. Metamorphic testing for (graphics) compilers. In *Proceedings of the 1st International Workshop on Metamorphic Testing, MET@ICSE 2016, Austin, Texas, USA, May 16, 2016*, pages 44–47. ACM, 2016. doi:10.1145/2896971.2896978.
- 17 Google. Amber GitHub repository, 2020. URL: <https://github.com/google/amber>.
- 18 Google. SwiftShader GitHub repository, 2020. URL: <https://github.com/google/SwiftShader>.
- 19 GPUOpen Drivers. LLVM-based pipeline compiler GitHub repository, 2020. URL: <https://github.com/GPUOpen-Driver/llpc>.
- 20 The Khronos Vulkan Working Group. *Vulkan 1.1.141 - A Specification (with all registered Vulkan extensions)*. The Khronos Group, 2019. URL: <https://www.khronos.org/registry/vulkan/specs/1.1-extensions/pdf/vkspec.pdf>.
- 21 K. V. Hanford. Automatic generation of test cases. *IBM Systems Journal*, 9:242–257, 1970.
- 22 John Kessenich, editor. *The OpenGL Shading Language Version 4.60.7*. The Khronos Group, 2019. URL: <https://www.khronos.org/registry/OpenGL/specs/gl/GLSLangSpec.4.60.pdf>.
- 23 John Kessenich, Boaz Ouriel, and Raun Krisch, editors. *SPIR-V Specification, Version 1.5, Revision 2, Unified*. The Khronos Group, 2019. URL: <https://www.khronos.org/registry/spir-v/specs/unified1/SPIRV.pdf>.
- 24 Khronos Group. glslang GitHub repository, 2020. URL: <https://github.com/KhronosGroup/glslang>.
- 25 Khronos Group. Khronos Vulkan, OpenGL, and OpenGL ES conformance tests GitHub repository, 2020. URL: <https://github.com/KhronosGroup/VK-GL-CTS>.
- 26 Khronos Group. MoltenVk GitHub repository, 2020. URL: <https://github.com/KhronosGroup/MoltenVK>.
- 27 Khronos Group. SPIR-V Tools GitHub repository, 2020. URL: <https://github.com/KhronosGroup/SPIRV-Tools>.
- 28 Khronos Group. SPIRV-Cross GitHub repository, 2020. URL: <https://github.com/KhronosGroup/SPIRV-Cross>.
- 29 Jeffery Kline. Properties of the d-dimensional earth mover’s problem. *Discrete Applied Mathematics*, 265:128–141, 2019. doi:10.1016/j.dam.2019.02.042.
- 30 Alexander S. Kossatchev and Mikhail Posypkin. Survey of compiler testing methods. *Programming and Computer Software*, 31(1):10–19, 2005. doi:10.1007/s11086-005-0008-6.
- 31 Vu Le, Mehrdad Afshari, and Zhendong Su. Compiler validation via equivalence modulo inputs. 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*, pages 216–226. ACM, 2014. doi:10.1145/2594291.2594334.
- 32 Jon Leech, editor. *OpenGL ES Version 3.2*. The Khronos Group, 2019. URL: https://www.khronos.org/registry/OpenGL/specs/es/3.2/es_spec_3.2.pdf.
- 33 Christopher Lidbury, Andrei Lascu, Nathan Chong, and Alastair F. Donaldson. Many-core compiler fuzzing. In David Grove and Steve Blackburn, editors, *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15-17, 2015*, pages 65–76. ACM, 2015. doi:10.1145/2737924.2737986.
- 34 David R. MacIver and Alastair F. Donaldson. Test-case reduction via test-case generation: Insights from the hypothesis reducer. In *34th European Conference on Object-Oriented Programming, ECOOP 2020*, volume 166 of *LIPICs*, pages 13:1–13:28. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020.
- 35 Michaël Marcozzi, Qiye Tang, Alastair F. Donaldson, and Cristian Cadar. Compiler fuzzing: how much does it matter? *PACMPL*, 3(OOPSLA):155:1–155:29, 2019. doi:10.1145/3360581.
- 36 William M. McKeeman. Differential testing for software. *Digital Technical Journal*, 10(1):100–107, 1998. URL: <http://www.hpl.hp.com/hpjournal/dtj/vol10num1/vol10num1art9.pdf>.
- 37 Microsoft. DirectX shader compiler GitHub repository, 2020. URL: <https://github.com/microsoft/DirectXShaderCompiler>.

- 38 NVIDIA. Security bulletin: Nvidia gpu display driver contains multiple vulnerabilities in the kernel mode layer handler, 2018. , see “NVIDIA GPU Display Driver contains a vulnerability in the kernel mode layer handler where an incorrect detection and recovery from an invalid state produced by specific user actions may lead to a denial of service”. URL: https://nvidia.custhelp.com/app/answers/detail/a_id/4525/.
- 39 Moritz Pflanzner, Alastair F. Donaldson, and Andrei Lascu. Automatic test case reduction for opencl. In *Proceedings of the 4th International Workshop on OpenCL, IWOCL 2016, Vienna, Austria, April 19-21, 2016*, pages 1:1–1:12. ACM, 2016. doi:10.1145/2909437.2909439.
- 40 John Regehr, Yang Chen, Pascal Cuoq, Eric Eide, Chucky Ellison, and Xuejun Yang. Test-case reduction for C compiler bugs. In Jan Vitek, Haibo Lin, and Frank Tip, editors, *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '12, Beijing, China - June 11 - 16, 2012*, pages 335–346. ACM, 2012. doi:10.1145/2254064.2254104.
- 41 Mark Segal and Kurt Akeley, editors. *The OpenGL Graphics System: A Specification Version 4.6 (Core Profile)*. The Khronos Group, 2019. URL: <https://www.khronos.org/registry/OpenGL/specs/gl/glspec46.core.pdf>.
- 42 Sergio Segura, Gordon Fraser, Ana B. Sánchez, and Antonio Ruiz Cortés. A survey on metamorphic testing. *IEEE Trans. Software Eng.*, 42(9):805–824, 2016. doi:10.1109/TSE.2016.2532875.
- 43 Robert J. Simpson and John Kessenich, editors. *The OpenGL ES Shading Language Version 3.20.6*. The Khronos Group, 2019. URL: https://www.khronos.org/registry/OpenGL/specs/es/3.2/GLSL_ES_Specification_3.20.pdf.
- 44 Chengnian Sun, Vu Le, and Zhendong Su. Finding compiler bugs via live code mutation. 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, part of SPLASH 2016, Amsterdam, The Netherlands, October 30 - November 4, 2016*, pages 849–863. ACM, 2016. doi:10.1145/2983990.2984038.
- 45 Chengnian Sun, Vu Le, Qirun Zhang, and Zhendong Su. Toward understanding compiler bugs in GCC and LLVM. In Andreas Zeller and Abhik Roychoudhury, editors, *Proceedings of the 25th International Symposium on Software Testing and Analysis, ISSTA 2016, Saarbrücken, Germany, July 18-20, 2016*, pages 294–305. ACM, 2016. doi:10.1145/2931037.2931074.
- 46 Qiuming Tao, Wei Wu, Chen Zhao, and Wuwei Shen. An automatic testing approach for compiler based on metamorphic testing technique. In Jun Han and Tran Dan Thu, editors, *17th Asia Pacific Software Engineering Conference, APSEC 2010, Sydney, Australia, November 30 - December 3, 2010*, pages 270–279. IEEE Computer Society, 2010. doi:10.1109/APSEC.2010.39.
- 47 Stéfan van der Walt, Johannes L. Schönberger, Juan Nunez-Iglesias, François Boulogne, Joshua D. Warner, Neil Yager, Emmanuelle Gouillart, Tony Yu, and the scikit-image contributors. scikit-image: image processing in Python. *PeerJ*, 2:e453, June 2014. doi:10.7717/peerj.453.
- 48 Brian A. Wichmann. Some remarks about random testing, 1998. Available online at <https://www.semanticscholar.org/paper/Some-Remarks-about-Random-Testing-Wichmann/2ad3c4c2e1b0b5867a1aa3e7c2de4a17d9facead>.
- 49 Brian A. Wichmann and Z. J. Ciechanowicz, editors. *Pascal Compiler Validation*. John Wiley & Sons, Inc., 1983.
- 50 Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. Finding and understanding bugs in C compilers. In Mary W. Hall and David A. Padua, editors, *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011, San Jose, CA, USA, June 4-8, 2011*, pages 283–294. ACM, 2011. doi:10.1145/1993498.1993532.
- 51 Andreas Zeller and Ralf Hildebrandt. Simplifying and isolating failure-inducing input. *IEEE Trans. Software Eng.*, 28(2):183–200, 2002. doi:10.1109/32.988498.

Lifting Sequential Effects to Control Operators

Colin S. Gordon 

Drexel University, Philadelphia, PA, USA

<https://cs.drexel.edu/~csgordon>

csgordon@drexel.edu

Abstract

Sequential effect systems are a class of effect system that exploits information about program order, rather than discarding it as traditional commutative effect systems do. This extra expressive power allows effect systems to reason about behavior over time, capturing properties such as atomicity, unstructured lock ownership, or even general safety properties. While we now understand the essential denotational (categorical) models fairly well, application of these ideas to real software is hampered by the variety of source level control flow constructs and control operators in real languages.

We address this new problem by appeal to a classic idea: macro-expression of commonly-used programming constructs in terms of control operators. We give an effect system for a subset of Racket’s tagged delimited control operators, as a lifting of an effect system for a language without direct control operators. This gives the first account of sequential effects in the presence of general control operators. Using this system, we also re-derive the sequential effect system rules for control flow constructs previously shown sound directly, and derive sequential effect rules for new constructs not previously studied in the context of source-level sequential effect systems. This offers a way to directly extend source-level support for sequential effect systems to real programming languages.

2012 ACM Subject Classification Theory of computation → Semantics and reasoning; Theory of computation → Type structures

Keywords and phrases Type systems, effect systems, quantales, control operators, delimited continuations

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2020.23

Related Version An extended version of this paper containing full proofs is available [32] on arXiv at <https://arxiv.org/abs/1811.12285>.

1 Introduction

Effect systems extend type systems to reason about not only the shape of data, and available operations – roughly, what a computation produces given certain inputs – but to also reason about *how* the computation produces its result. Examples include ensuring data race freedom by reasoning about what locks a computation assumes held during its execution [1, 23, 11, 10], restricting sensitive actions (like UI updates) to dedicated threads [33], ensuring deadlock freedom [24, 34, 1, 69], checking safe region-based memory management [72, 50], or most commonly checking that a computation handles (or at least indicates) all errors it may encounter – Java’s checked exceptions [35] are the most widely used effect system.

Most effect systems discard information about program order: the same join operation on a join semilattice of effects is used to overapproximate different branches of a conditional or different subexpressions executed in sequence. Despite this simplicity, these traditional *commutative* effect systems (where the combination of effects is always a commutative operation) are powerful. Still, many program properties of interest are sensitive to evaluation order. For example, commutative effect systems handle scoped **synchronized** blocks as in Java with ease: the effect of (the set of locks required by) the **synchronized**’s body is permitted to contain the synchronized lock, in addition to the locks required by the overall construct.



© Colin S. Gordon;

licensed under Creative Commons License CC-BY

34th European Conference on Object-Oriented Programming (ECOOP 2020).

Editors: Robert Hirschfeld and Tobias Pape; Article No. 23; pp. 23:1–23:30

Leibniz International Proceedings in Informatics



LIPICs Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

But to support explicit lock acquisition and release operations that are not block-structured, an effect system must track whether a given expression acquires and/or releases locks, and must distinguish their ordering: releasing and then acquiring a given lock is not the same as acquiring before releasing. To this end, *sequential* effect systems (so named by Tate [70]) reason about effects with knowledge of the program’s evaluation order.

Sequential effect systems are much more powerful than commutative effect systems, with examples extending through generic reasoning about program traces [68, 67] and even propagation of *liveness* properties from oracles [45] – well beyond what most type systems support. The literature includes sequential effect systems for deadlock freedom [34, 69, 1, 10], atomicity [25, 26], trace-based security properties [68, 67], safety of concurrent communication [4, 57], general linear temporal properties with a liveness oracle [45], and more. Yet for all the power of this approach, for years each of the many examples of sequential effect systems in the literature individually rederived much structure common to all sequential effect systems. Recent years have seen efforts to unify understanding of sequential effect systems with general frameworks, first denotationally [70, 42, 55, 6], and recently as an extension to the join semilattice model [30]. These frameworks can describe the structure of established sequential effect systems from the literature.

However, these generic frameworks stop short of what is necessary to apply sequential effect systems to real languages: they lack generic treatments of critical features of real languages that interact with evaluation order – control operators, including established features like exceptions and increasingly common features like generators [14]. And with the exception of the effect systems used to track correct return types with delimited continuations (answer type modification [16, 5, 44]), there are no sequential effect systems that consider the interaction of control and sequential effects. This means *promising sequential effect systems [68, 67, 45, 25, 34, 4, 57, 69, 10] cannot currently be applied directly to real languages like Java [35], Racket, C# [52], Python [71], or JavaScript [54].*

Control operators effectively reorder, drop, or duplicate portions of a program’s execution at runtime, changing evaluation order. In order to reason precisely about flexible rearrangement of evaluation order, a sequential effect system must reason about control operators. The classic example is again Java’s `try-catch`: if the body of a `try` block both acquires and releases a lock this is good, but if an exception is thrown mid-block the release may need to be handled in the corresponding catch. Clearly, applying sequential effect systems to real software requires support for exceptions in a sequential effect system. Working out just those rules is tempting, but exceptions interact with loops. The effect before a throw inside a loop – which a catch block may need to “complete” (e.g., by releasing a lock) – depends on whether the throw occurs on the first or *n*th iteration. Many languages include more than simply `try-catch`, for example with the *generators* (a form of coroutine) now found in C# [52], Python [71], and JavaScript [54]. These interact with exceptions *and* loops. Treating each new control operator individually seems inefficient.

An alternative to studying all possible combinations of individual control constructs in common languages is to study more general constructs, such as the very general delimited continuations [22, 19] present in Racket. These are useful in their own right (for Racket, or the project to add them to Java [36]), and can macro-express many control flow constructs and control operators of interest, including loops, exceptions [27], coroutines [39, 40], generators [14], and more [15]. Then general principles can be derived for the general constructs, which can then be applied to or specialized for the constructs of interest. This both solves the open question of how to treat general control operators with sequential effect systems, and leads to a basis for more compositional treatment of loops, exceptions, generators, and future additions to languages. This is the avenue we pursue in this paper.

Delimited continuations solve the generality problem, but introduce new challenges since sequential effect systems can track evaluation order [68, 67, 31, 45]. The effect of an expression that aborts out of a prompt depends on what was executed before the abort, but not after. The body of a continuation capture (`call/cc`) must be typed knowing the effect of the *enclosing context* – the code executing after, but not before (up to the enclosing prompt). We lay the groundwork for handling modern control operators in a sequential effect system:

- We give the first generic characterization of sequential effects for continuations, by giving a *generic* lifting of a control-unaware sequential effect system into one that can support tagged delimited continuations. The construction we describe provides a way to automatically extend existing systems with support for these constructs, and likewise will permit future sequential effect system designers to ignore control operations initially and add support later for free (by applying our construction). As a consequence, we can transfer prior sequential effect systems designed *without* control operators to a setting *with* control operators.
- We give sequential effect system rules for `while` loops, `try-catch`, and generators by deriving them from their macro-expression [20] in terms of more primitive operators. The loop characterization was previously known (and technically a control flow construct, not a general control operator), but was given as primitive. The others are new to our work, and necessary developments in order to apply sequential effect systems to most modern programming languages. The derivation approach we describe can be applied to other control operators that are not explicitly treated in the paper.
- We demonstrate how prior work’s notion of an iteration operator [30, 31] derived from a closure operator on the underlying effect lattice is not specific to loops, but rather provides a general tool for solving recursive constraints in sequential effect systems.
- We prove syntactic type safety for a type system using our sequential control effect transformation with any underlying effect system.

2 Background

We briefly recall the details of sequential type-and-effect systems, and tagged delimited continuations. We emphasize the view of effect systems in terms of a *control flow algebra* [55] – an algebraic structure with operations corresponding to the ways an effect system might combine the effects from subexpressions in a program.

2.1 Sequential Effect Systems

Traditional type-and-effect systems extend the typing judgment $\Gamma \vdash e : \tau$ for an additional component. The extended judgment form $\Gamma \vdash e : \tau \mid \chi$ is read “under local variable assumptions Γ , the expression e evaluates to a value of type τ (or diverges), with effect χ during evaluation.” The last clause of that reading is vague, but carries specific meanings for specific effect systems. For checked exceptions, it could be replaced by “possibly throwing exceptions χ during evaluation” where χ would be a set of checked exceptions. For a data race freedom type system reasoning about lock ownership, it could be replaced by “and is data race free if executed while locks χ are held.”

The join semilattice structure of standard effect systems is well-known, as are the corresponding denotational analogues (e.g., indexed monads [74]). The limitation common to all of these systems, however, is that they discard program order, using the (commutative) join for any combination of effects. In contrast, there is growing work on *sequential* [70] effect systems, which capture a wide array of order-sensitive phenomena. This includes

23:4 Lifting Sequential Effects to Control Operators

effect systems for atomicity [25, 26], deadlock freedom [69, 34, 10, 1], race freedom with explicit lock acquisition and release [69, 30], message passing concurrency safety [57, 4], security checks [68], and (with the aid of an oracle for liveness properties) general linear-time properties [45]. Tate labels these systems *sequential* effect systems [70], as their distinguishing feature is the use of an additional sequencing operator to join effects where one is known to be evaluated before another. Consider the sequential rules for functions, function application, conditionals, and while loops:

$$\begin{array}{c}
 \text{T-APP} \\
 \frac{\Gamma \vdash e_1 : \tau \xrightarrow{\chi} \sigma \mid \chi_1 \quad \Gamma \vdash e_2 : \tau \mid \chi_2}{\Gamma \vdash e_1 e_2 : \sigma \mid \chi_1 \triangleright \chi_2 \triangleright \chi}
 \end{array}
 \qquad
 \begin{array}{c}
 \text{T-WHILE} \\
 \frac{\Gamma \vdash e_c : \text{boolean} \mid \chi_c \quad \Gamma \vdash e_b : \tau \mid \chi_b}{\Gamma \vdash \text{while } e_c e_b : \text{unit} \mid \chi_c \triangleright (\chi_b \triangleright \chi_c)^*}
 \end{array}$$

$$\begin{array}{c}
 \text{T-LAMBDA} \\
 \frac{\Gamma, x : \tau \vdash e : \sigma \mid \chi}{\Gamma \vdash (\lambda x. e) : \tau \xrightarrow{\chi} \sigma \mid I}
 \end{array}
 \qquad
 \begin{array}{c}
 \text{T-IF} \\
 \frac{\Gamma \vdash e_c : \text{bool} \mid \chi_c \quad \Gamma \vdash e_t : \tau \mid \chi_t \quad \Gamma \vdash e_f : \tau \mid \chi_f}{\Gamma \vdash \text{if } e_c e_t e_f : \tau \mid \chi_c \triangleright (\chi_t \sqcup \chi_f)}
 \end{array}$$

The sequencing operator \triangleright is associative but *not* (necessarily) commutative. Thus the effect in the new T-APP reflects left-to-right evaluation order: first the function position is reduced to a value, then the argument, and then the function body is executed. The conditional rule reflects the execution of the condition followed by *either* (via commutative join) the true or false branch. The while loop uses an iteration operator $(-)^*$ to represent 0 or more repetitions of its argument; we will return to its details later. The effect of T-WHILE reflects the fact that the condition will always be executed, followed by 0 or more repetitions of the loop body and checking the loop condition again. The rule for typing lambda expressions switches from a bottom element, to a general unit effect: identity for sequential composition.

To formalize the intuition above, Gordon [30] proposed *effect quantales* as a model that captures prior effect systems' structure:

► **Definition 1** (Effect Quantale). *An effect quantale is a join-semilattice-ordered monoid with nilpotent top. That is, it is a structure $(E, \sqcup, \top, \triangleright, I)$ where:*

- (E, \sqcup, \top) is an upper-bounded join semilattice
- (E, \triangleright, I) is a monoid
- \top is nilpotent for sequencing ($\forall x. x \triangleright \top = \top = \top \triangleright x$)
- \triangleright distributes over \sqcup on both sides: $a \triangleright (b \sqcup c) = (a \triangleright b) \sqcup (a \triangleright c)$ and $(a \sqcup b) \triangleright c = (a \triangleright c) \sqcup (b \triangleright c)$

The structure extends a join semilattice with a sequencing operator, a designated error element to model possibly-undefined combinations, and laws specifying how the operators interact. Top (\top) is used as an indication of a type error, for modeling partial join or sequence operators: expressions with effect \top are rejected. \sqcup is used to model non-deterministic joins (e.g., for branches) as in the commutative systems, and \triangleright is used for sequencing. The default effect of “uninteresting” program expressions (including values) becomes the unit I rather than a bottom element (which need not exist). As a consequence of the distributivity laws, it follows that \triangleright is also monotone in both arguments, for the standard partial order derived from a join semilattice: $x \sqsubseteq y \equiv x \sqcup y = y$.

Gordon [30] also showed how to exploit *closure operators* [8, 9, 29] to impose a well-behaved notion of iteration (the $(-)^*$ operator from T-WHILE) that coincides with manually-derived versions for the effect quantales modeling prior work for many effect quantales. Gordon [31] recently generalized the construction, and showed that large general classes of effect quantales meet the criteria to have such an iteration operator. The effect quantales for which the generalized iteration is defined are called *laxly iterable*. An effect quantale is *laxly iterable* if for every element x , the set of subidempotent elements ($\{s \mid s \triangleright s \sqsubseteq s\}$) greater than both x and I has a least element. This is true of all known effect quantales corresponding to systems in the literature.

The iteration operator for an iterable effect quantale takes each effect x to the least subidempotent effect greater than or equal to $x \sqcup I$ (which exists, by the definition of laxly iterable). This iteration operator satisfies 5 essential properties for any notion of iteration [31], which we will find useful when deriving rules for loops. Iteration operators are *extensive* ($\forall e. e \sqsubseteq e^*$), *idempotent* ($\forall e. (e^*)^* = e^*$), *monotone* ($\forall e, f. e \sqsubseteq f \Rightarrow e^* \sqsubseteq f^*$), *foldable* ($\forall e. e \triangleright e^* \sqsubseteq e^*$ and $e^* \triangleright e \sqsubseteq e^*$), and *possibly-empty* ($\forall e. I \sqsubseteq e^*$). Another useful property of iteration that we will sometimes use is that $\forall x, y. x^* \sqcup y^* \sqsubseteq (x \sqcup y)^*$. Gordon [30, 31] gives more details on closure operators and the derivation of iteration. We merely require its existence and properties.

For our intended goal of giving a transformation of any arbitrary sequential effect system into one that can use tagged delimited continuations, we require *some* abstract characterization. We choose effect quantales as the abstraction for lifting for several reasons. First, they characterize the structure of a range of concrete systems from prior work [30, 31], while other proposals omit structure that is important to these concrete systems. Second, while effect quantales are not maximally general, they remain very general: the motivating example for Tate’s work [70] (which *is* maximally general) can be modeled as an effect quantale. Third, we would like to check whether our derived rules are sensible; effect quantales are the only abstract characterization for which imperative loops have been investigated, offering appropriate points of comparison. Finally, the iteration construction on effect quantales offers a natural approach to solving recursive constraints on effects, which we will use in deriving closed-form derived rules for macro-expressed control flow constructs and control operators.

As a running example throughout the paper, we will use a simplification of various trace or history effect systems [68, 67, 45]. For a set (alphabet) of events Σ , consider the non-empty subsets of Σ^* – the set of possibly-empty strings of letters drawn from Σ (the strings, not the subsets, may be empty). This gives an effect quantale $\mathcal{T}(\Sigma)$ whose elements are these subsets or an additional top-most error element Err . Join is simply set union lifted to propagate Err . Sequencing is the double-lifting of concatenation, first to sets ($A \cdot B = \{xy \mid x \in A \wedge y \in B\}$), then again to propagate Err . The unit for sequencing is the singleton set of the empty string, $\{\epsilon\}$. If Σ is a set of events of interest – e.g., security events – then effects drawn from this effect quantale represent sets of possible finite event sequences executed by a program. Effects drawn from this effect quantale show the possible sequences of operations code may execute, which will allow us to show explicitly how fragments of program execution are rearranged when using control operators. For our examples, we will assume a family of language primitives $\text{event}[\alpha]$ with effect $(\emptyset, \emptyset, \{\alpha\})$ (similar to Koskinen and Terauchi [45]), where α is drawn from a set Σ of possible events. The key challenge we face in this paper is, viewed through the lens of $\mathcal{T}(\Sigma)$, to ensure that when continuations are used, the effect system does not lose track of events of interest or falsely claim a critical event occurs where it may not.

2.2 Tagged Delimited Continuations

Control operators have a long and rich history, reaching far beyond what we discuss here. Many different control operators exist, and many are macro-expressible [20] in terms of each other (i.e., can be translated by direct syntactic transformation into another operator), though some of these translations require the assumption of mutable state, for example. But a priori there is no single most general construct to study which obviously yields insight on the source-level effect typing of other constructs. A suitable starting place, then, is to target a highly expressive set of operators that see use in a real language. If the operators

$$\begin{aligned}
E ::= & \bullet \mid (E \ e) \mid (v \ E) \mid (\% \ t \ E \ v) \mid (\text{call/cc } t \ E) \mid (\text{call/comp } t \ E) \mid (\text{abort } t \ e) \\
& \boxed{\sigma; e \xrightarrow{q} \sigma; e} \quad \text{E-APP} \frac{}{\sigma; ((\lambda x. e) \ v) \xrightarrow{I} \sigma; e[v/x]} \quad \text{E-PROMPTVAL} \frac{}{\sigma; (\% \ell \ v \ h) \xrightarrow{I} \sigma; v} \\
& \boxed{\sigma; e \xrightarrow{q} \sigma; e} \quad \text{E-CONTEXT} \frac{\sigma; e \xrightarrow{q} \sigma'; e'}{\sigma; E[e] \xrightarrow{q} \sigma'; E[e']} \quad \text{E-ABORT} \frac{E' \text{ contains no prompts for } \ell}{\sigma; E[(\% \ell \ E'[(\text{abort } \ell \ v))] \ h)] \xrightarrow{I} \sigma; E[h \ v]} \\
& \text{E-CALLCC} \frac{E' \text{ contains no prompts for } \ell}{\sigma; E[(\% \ell \ E'[(\text{call/cc } \ell \ k))] \ h)] \xrightarrow{I} \sigma; E[(\% \ell \ E'[(k \ (\text{cont } \ell \ E'))]] \ h)} \\
& \text{E-VOKEECC} \frac{E' \text{ contains no prompts for } \ell}{\sigma; E[(\% \ell \ E'[(\text{cont } \ell \ E'') \ v])] \ h)] \xrightarrow{I} \sigma; E[(\% \ell \ E''[v] \ h)]
\end{aligned}$$

■ **Figure 1** Operational semantics.

are sufficiently expressive, this provides not only a sequential type system for an expressive source language directly, but also supports deriving type rules for *other* languages' control constructs, based on their macro-expression in terms of the studied control operators.

We study a subset of the tagged delimited control operators [22, 19, 64, 65, 66] present in Racket [27], shown in Figure 1. The semantics include both local (\rightarrow) and global (\Rightarrow) reductions on configurations consisting of a state σ and expression e . All continuations in Racket are delimited, and tagged. There is a form of *prompt* that limits the scope of any continuation capture: $(\% \ \text{tag} \ e \ e2)$ is a tagged prompt with tag **tag**, body **e**, and abort handler **e2**. Without tags, different uses of continuations – e.g., error handling or concurrency abstractions – can interfere with each other [64]; as a small example, if loops and exceptions were both implemented with *undelimited* continuations, throwing an exception from inside a loop inside a try-catch would jump to the loop boundary, not the catch. Thus prompts, the continuation-capturing primitives **call/cc** and **call/comp**, and the **abort** primitive all specify a tag, and only prompts with the specified tag are used to interpret continuation and abort boundaries. This permits jumping over unrelated prompts (e.g., so exceptions find the nearest *catch*, not merely the nearest control construct). In most presentations of delimited continuations, tags are ignored (equivalently, all tags are equal), while most implementations retain them for the reasons above. Here the tags are essential to the theory as well: an abort that “skips” a different prompt must be handled differently by our type-and-effect system.

call/cc tag f is the standard (delimited) call-with-current-continuation: **f** is invoked with a delimited continuation representing the current continuation up to the nearest prompt with tag **tag** (E-CALLCC). Invoking that continuation (E-VOKEECC) replaces the context up to the nearest dynamically enclosing prompt with the same tag, leaving the delimiting prompt in place. Both capture and replacement are bounded by the nearest enclosing prompt for the specified tag. The surrounding captured or replaced context (E' in both rules) may contain prompts for other tags, but not the specified tag. Racket also includes **(abort t e)** (absent in many formalizations of continuations), which evaluates **e** to a value, then replaces the enclosing prompt (of the specified tag **t**) with an invocation of the handler applied to that value (E-ABORT). Racket's rules differ from some uses of **abort** in the literature. Figure 1's rules are Flatt et al.'s rules [27] without continuation marks and **dynamic-wind**. Flatt et al. formalized Racket's control operators in Redex [21, 43], including showing they passed the Racket implementation tests for those features. We have verified the rules above continue to pass the relevant tests in Redex (see supplementary material [32]).

We chose this set of primitives, over related control operators [63] such as `shift/reset` or `shift0/reset0` which can simulate these primitives, for several reasons. First, they are general enough to use for deriving rules for higher-level constructs like generators from their macro-expansion. Second, the control operators we study are implemented as primitives in a real, mature language implementation (Racket), used in real software [46]. And finally, it is known [27] how these control operators interact with other useful control operators like `dynamic-wind` [38] (relevant to `finally` or `synchronized` blocks) and continuation marks [13]. Thus our Racket subset is a suitable basis for future extension, while we are unaware of established extensions of `shift/reset`, `shift0/reset0`, etc. with continuation marks or `dynamic-wind`.

The operators we study can express loops, exceptions, coroutines [39, 40], and generators [14]. Racket also includes *compositional* continuations, whose application extends the current context rather than discarding it, giving completeness with respect to some denotational models [66], and alleviating space problems when using `call/cc` to simulate other families of control operators (it is known to macro-express another popular form of delimited continuations, the combination of `shift` and `reset` [27]). Our technical report [32] extends our development to include compositional continuations as well.

One final point about the semantics worth noting is the presence of effect annotations on the reduction arrows. These semantics are further adapted from Flatt et al. [27] to “emit” the primitive effect of the reduction, which is typical of syntactic type safety proofs for effect systems, including ours (Section 6). They do not influence evaluation, but only mark a relationship between the reduction rules and static effects. Non-unit (non-*I* effects) arise from a choice of primitives that depends on the particular effect system studied.

3 Growing Sequential Effects: Control, Prophecies, and Blocking

To build intuition for our eventual technical solution, we motivate its components through a series of progressively more sophisticated use cases. We will use $\mathcal{T}(\Sigma)$ in all of our examples, because we find traces to be an effective way of explaining the difficulties the effect system must address related to program fragments (i.e., events) being repeated, skipped, or reordered. A reader may choose to impart security-specific meanings to these events (as Skalka et al. [68] do) or as any other protocol of personal interest (e.g., lock acquisition and release). However, our development in Section 4 is *not* specific to this effect quantale, but instead parameterized over an arbitrary effect quantale. Our goal is to develop a sequential effect system based on transforming an *underlying* base effect quantale Q into a structure we will call $\mathcal{C}(Q)$ with sequencing and join operations, and a unit effect. This ensures our transformation works for *any* valid effect quantale, which includes all sequential effect systems we are aware of [30, 31].

► **Use Case 1 (Control-Free Programs).** Since programs are not required to use control operators, our solution must include a restriction equivalent to the class of underlying effects to reason about. For example, if `event[α]` has effect $\{\alpha\} \in \mathcal{T}(\Sigma)$ and `event[β]` has effect $\{\beta\} \in \mathcal{T}(\Sigma)$, we should expect the effect of `event[α]; event[β]` to be somehow equivalent to sequencing those underlying effects – $\{\alpha\} \triangleright \{\beta\} = \{\alpha\beta\}$. This suggests underlying effects should be at least a component of continuation-aware effects.

► **Use Case 2 (Aborting Effects).** The simplest control behavior we can use is to abort to a handler, and this interacts with both sequencing and conditionals. Consider:

```
(% t ((if c (event[ $\alpha$ ]) (abort t 3)); event[ $\beta$ ]) ( $\lambda$ n. event[ $\gamma$ ]) )
```

Assuming c is a variable (i.e., pure), there are two paths through this term:

- If c is true, the code will emit events α and β (in order), and not execute the handler
- If c is false, the code will abort to the handler, which will emit event γ .

So intuitively, the effect for this term should contain those traces; ideally the effect would be $\{\alpha\beta, \gamma\}$, containing only those traces. For an effect system to validate this effect for this term, it must not only track ordinary underlying effects, but also two aspects of the `abort` operation's behavior: it causes some code to be discarded (the `event[\beta]`), and it causes the handler to run. We can track this information by making effects pairs of two components: a set of behaviors up to an abort (which we will call the *control effect set*, since it tracks effects due to non-local control transfer), and an underlying effect for when no abort occurs. We could then give the body of the prompt the effect $(\{\text{abort}(\{\epsilon\})\}, \{\alpha\beta\})$ to indicate that it either executes normally producing a trace $\alpha\beta$, or it aborts to the nearest handler after doing nothing (more precisely, after performing actions with the unit effect for $\mathcal{T}(\Sigma)$). The type rule for prompts can then recognize the body may abort, and for each possible prefix of an aborting execution, add into the overall (underlying) effect of the prompt the result of sequencing that prefix with the *handler's* effect: here, $(\{\epsilon\} \triangleright \{\gamma\}) \sqcup \{\alpha\beta\} = \{\alpha\beta, \gamma\}$.

Above the body effect was given as a whole from intuition, but in general this must be built compositionally from subexpression effects, motivating further questions. First, what is the effect of the subterm `abort t 3`? In particular, what is its underlying effect? One sound choice would be $I(\{\epsilon\}$ for $\mathcal{T}(\Sigma)$), but this would introduce imprecision: it would produce effects suggesting it was possible to execute only event β (since the conditional's underlying effect would include both α and the empty trace ϵ , sequenced with β). Instead, we will make the underlying component *optional*, writing \perp when it is absent. We will continue to use metavariables Q to indicate a definitely present element of the underlying effect quantale, but will use the convention of underlining metavariables (e.g., \underline{Q}) when they may be \perp . This permits the conditional's underlying effect to only contain the trace α , because joining the branches' effects can simply ignore the missing underlying effect from the aborting branch.

Second, we should consider how the sequencing and join operations interact with abort effects. While component-wise union/join is a natural (and working) starting point, sequencing is less obvious. Prefixing the last example's body with an extra event is instructive:

```
(% t (event[\delta]; (if c (event[\alpha]) (abort t 3)); event[\beta]) (\lambda n. event[\gamma]) )
```

Execution could generate two traces: $\delta\alpha\beta$ and $\delta\gamma$. So *both* traces in the effect for the last body should gain this δ prefix: not only the underlying effect component, but also the portion related to the `abort`. This suggests the following definition of sequencing:

$$(C_1, Q_1) \triangleright (C_2, Q_2) = (C_1 \cup (Q_1 \triangleright C_2), Q_1 \triangleright Q_2)$$

Assuming there is a lifting (given later) of the underlying sequencing to possibly-absent underlying effects ($Q_1 \triangleright Q_2$), this reflects the natural ways of combining paths through these terms: the *control effect set* collecting abort behaviors should include both the abort behaviors from C_1 (an execution corresponding to one of those behaviors means nothing from the second effect will execute), as well as the result of first executing *normal* behaviors of the first effect (e.g., $\{\delta\}$) before the aborting behaviors of the second – $Q_1 \triangleright C_2$. So the effect of this new example's prompt body should be the join of the two branches $((\{\text{abort}(\{\epsilon\})\}, \perp) \sqcup (\emptyset, \{\alpha\}) = (\{\text{abort}(\{\epsilon\})\}, \{\alpha\}))$, sequenced between the effects of event δ and event β : $(\emptyset, \{\delta\}) \triangleright (\{\text{abort}(\{\epsilon\})\}, \{\alpha\}) \triangleright (\emptyset, \{\beta\}) = (\emptyset, \{\delta\}) \triangleright (\{\text{abort}(\{\epsilon\})\}, \{\alpha\beta\}) = (\{\text{abort}(\{\delta\})\}, \{\delta\alpha\beta\})$. Repeating our informal prompt handling above gives us the new expected underlying effect $\{\delta\alpha\beta, \delta\gamma\}$. We refer to the way the control effect set accumulates prefixes on the left as *left-accumulation* of effects. This definition is associative, and distributes

over the component-wise union/join. For now we continue with the simplifying assumption that all tags are τ (equivalent to *untagged* continuations), and consider issues with multiple tags in Use Case 6. A final detail deferred to Section 4 is that the value thrown by an `abort` must be of the type expected by the corresponding handler.

This is already part-way to our goal of deriving type rules for common control operators from delimited continuations: the work so far supports basic checked exceptions:

$$\llbracket \text{try } e \text{ catch } C \Rightarrow e_c \rrbracket = (\% C \llbracket e \rrbracket \llbracket e_c \rrbracket) \quad \llbracket \text{throw}_C e \rrbracket = (\text{abort } t_C \llbracket e \rrbracket)$$

The formal rules for prompts and aborts will directly dictate the rules for these macro-expressions of checked exceptions, just as the informal discussion here transfers directly to these macro-expressed checked exceptions.

► **Use Case 3 (Invoking Simple Continuations).** Operationally, invoking an existing continuation is similar to using `abort` in that it discards the surrounding context up to the nearest prompt. But unlike uses of `abort`, invoking a continuation does not cause handler execution – instead (by E-INVOKEC) the prompt and handler remain in place. For the type rule for prompts to treat this additional mechanism, effects must indicate that invocation may occur. We can do this by extending effects slightly: instead of C in the effect (C, Q) only containing prefixes of aborting computations, it should also include prefixes of continuation invocations – this is why elements of C were labeled `abort(-)` in the previous example, and we can now include effects tagged by `replace(-)` to indicate control behaviors that *replace* the current continuation with a new one. Considering a term invoking a continuation k :

```
(%  $\tau$  (event  $[\delta]$ ; (if  $c$  (event  $[\alpha]$ ) (k ()); event  $[\beta]$ ) ( $\lambda n$ . event  $[\gamma]$ )) )
```

As in the previous example, one possible trace of this program is $\delta\alpha\beta$ when c is true. When c is false, however, this program will emit event δ , then invoke k , replacing the rest of the body with k 's captured continuation (with unit in its hole). Thus typing this requires also knowing additional information about k , not required in the `abort` case. We require continuations to carry a latent effect similar to functions: while the latent effect of a function $(\lambda x.e)$ describes the effect of the term obtained by substituting an appropriately-typed value v into e – the effect of $e[v/x]$ – the latent effect of a continuation $(\text{cont}^\tau \ell E)$ describes the effect of plugging an appropriately-typed value v into E 's hole – the effect of $E[v]$.

Assuming k has only underlying effects – e.g., if $k = (\text{cont}^{\text{unit}} t (\bullet; \text{event}[\eta]))$, then we should expect the type rule for invocation to take that underlying effect from k 's latent effect and move it to the control effect set, under `replace(-)`. So if k 's latent effect is $(\emptyset, \{\eta\})$, the effect of $(k \ ())$ should be $(\{\text{replace}(\{\eta\})\}, \perp)$. Sequential composition should be extended to treat `replace` control effects similar to `abort` control effects, by accumulating on the left. With that adjustment, we can conclude the body above has effect $(\{\text{replace}(\{\delta\eta\})\}, \{\delta\alpha\beta\})$. The type rule for prompts can treat these similarly to `abort` control effects, but without sequencing them with the handler (which doesn't execute in this case), producing an overall effect $(\emptyset, \{\delta\alpha\beta, \delta\eta\})$. Notice that this is slightly different from `abort(-)` control effects: those track *only* the prefix effect before the abort, but the approach just outlined would have `replace(-)` control effects include prefix effects before invoking the continuation *and* the underlying effect of behaviors *after* the control operation. This corresponds to the location of the behavior that executes after the control operation – remote for aborts (the handler is non-local) and local for continuation invocation (the continuation is at the call site).

► **Use Case 4 (Invoking Continuations that Abort or Invoke Other Continuations).** The example above assumed k had only underlying effects, but in general k 's body might use `abort` or invoke other continuations. In such cases, k 's latent effect would be some pair (C, Q) for

23:10 Lifting Sequential Effects to Control Operators

non-empty control effect set C . It turns out simply treating those naïvely – including them into the control effect set for invoking k – is adequate for now (we revisit this when considering multiple tags). If $k = (\text{cont}^{\text{unit}} t (\bullet; \text{event}[\eta]; \text{abort } t \ 3))$ in the previous example, then the latent effect will be $(\{\text{abort}(\{\eta\})\}, \perp)$, and simply making the effect of $(k \ _)$ be the same (dropping the absent underlying latent effect since the continuation can only return via control operators, but making the application’s underlying effect \perp because by definition it does not return directly) gets the expected result at the prompt, including the trace $\delta\eta\gamma$ from emitting δ , invoking k , emitting η (from k ’s restored body) and aborting to the handler. Because the latent control effects from k ’s body already includes the prefixes from the start of k ’s body to the `abort`, the existing left accumulation in our definition of \triangleright correctly accumulates prefixes from the site of continuation invocation, into the continuation, to its uses of control operations. In general (for a single tag), invoking a continuation with latent effect (C, Q) has effect $(C \cup \{\text{replace}(Q)\}, \perp)$, though this assumes a non-empty underlying effect for the continuation – an assumption the final type rules will need to relax, along with extension for multiple tags, and issues with the continuation’s argument and result type.

► **Use Case 5 (Capturing Continuations).** Typing uses of `call/cc` is the most complex problem this effect system must address. For a term $(\text{call/cc } t \ (\lambda k. \ e))$, the rule must type the body function, which means choosing a type for the variable k that will be bound to the continuation. We defer continuation argument and result types for now. The latent effect of the continuation parameter depends on the effect of the code in the captured context – code that is (at runtime) “between” the `call/cc` and the (dynamically) nearest prompt. Consider applying a purely local type rule for `call/cc` to this simple example:

```
(% t ((call/cc t (λk. e)); (foo 3)) ...)
```

Here the context captured will clearly be $(\bullet; (\text{foo } 3))$. This is awkward when typing the subterm $(\text{call/cc } t \ (\lambda k. \ e))$, because that context is not a subterm of what is being typed.

We will take a “guess and check” approach¹ to typing `call/cc`, assuming a certain latent effect and ensuring it can be checked elsewhere when additional information is available (in our case, in the type rule for prompts), essentially a form of *prophecy* [2, 3]. We track prophecies in a third (final) component, the *prophecy set*, resulting in three-component effects (P, C, Q) . We call these three-part effects *continuation effects*, using metavariable χ . An individual prophecy records the assumed latent effect of a continuation. Since that continuation may capture further continuations, the prophecy must predict a full continuation effect, making prophecies and continuation effects mutually recursive.

Prophecies alone are only local guesses about non-local phenomena; the effect system requires a way to validate them. Intuitively, a prophecy that a `call/cc` captures a continuation with latent effect (P, C, Q) is valid if in any dynamic context the `call/cc` is evaluated, the effect of the program text “between” the capture and the nearest prompt has an effect less than (P, C, Q) in the partial order (to be defined). Checking this statically is non-trivial. While the full context captured is visible in $(\% t ((\text{call/cc } t \ (\lambda k. \ e)); \text{event}[\alpha]) (\lambda n. \ \dots))$, enclosing the `call/cc` in a function makes it non-local, and that function may be invoked in multiple contexts. Minor changes to the context may also (need to) break typability: if this `call/cc`’s body requires that the context has only underlying effect $\{\alpha\}$, then adding an additional `event` to the end of the prompt’s body must make the expression untypable.

¹ In the sense that a human constructing a typing derivation involving `call/cc` would need to informally guess a prophecy, then use the type rules to check that it was a sound guess.

We can again turn to the idea of accumulation, but *to the right*. Thus we write individual prophecies (again, ignoring tags and argument and result types of continuations) as `prophecy χ obs χ'` – indicating that for a certain `call/cc`, an effect of χ was prophesized, meaning that the term whose effect contains this prophecy assumed a latent effect χ for the continuation, and the effect of the term fragment between the `call/cc` and the boundary of the term has effect χ' , which we call the *observation*. We will extend sequential composition to accumulate effects to the right into the observation. When type checking a prompt, there is no more to accumulate because the prompt is the boundary of the continuation capture: at that point, the observation reflects the *actual* effect of the code *statically* on control flow paths between `call/cc` and the prompt, so the prophecy was sound if the observed effect is less than the prophecy effect.

With this prophecy-and-observation approach, let us consider:

```
(% t (((call/cc t (λk. e)); event[α]); event[β]) (λn. ...))
```

The type rule for `call/cc` can assume a surrounding context effect of $(\emptyset, \emptyset, \{\alpha\beta\})$. Let us assume for simplicity the inner body of the `call/cc` is pure (unit effect). Then the effect of the `call/cc` term can be $(\{\text{prophecy } (\emptyset, \emptyset, \{\alpha\beta\}) \text{ obs } (\emptyset, \emptyset, I)\}, \emptyset, I)$ (writing the unit effect as simply I for readability). Then the effect of sequencing that with the `event[α]` (i.e., $(\emptyset, \emptyset, \{\alpha\})$) should both update the underlying effect *and the observation* of the prophecy: $(\{\text{prophecy } (\emptyset, \emptyset, \{\alpha\beta\}) \text{ obs } (\emptyset, \emptyset, \{\alpha\})\}, \emptyset, \{\alpha\})$. Repeating this for the next event, we would get $(\{\text{prophecy } (\emptyset, \emptyset, \{\alpha\beta\}) \text{ obs } (\emptyset, \emptyset, \{\alpha\beta\})\}, \emptyset, \{\alpha\beta\})$. At that point, this has typed the entire body of the prompt, and the type rule for the prompt can check that the observed effect is no greater than the prophesized effect – in this case trivially since they are equal.

► **Use Case 6 (Trouble with Tags)**. Work with delimited continuations typically formalizes work without tags, but then adds them to the implementation as a “straightforward” extension. There are, however, several ways in which handling multiple tags is *non-trivial* in this context, warranting an explicit treatment. First, nested continuations result in more subtlety when handling control effect sets in the prompt rule. An obvious change is for a prompt tagged `t` to ignore aborts and continuation invocations (elements of the control effect set) that target a different tag (which requires tracking the target tag for each `replace` or `abort` effect). However, this is insufficient. A continuation `k` that restores up to tag t may include a latent abort to tag $t2$ – but if the nearest prompt is tagged $t2$, this is subtle. Consider the continuation $k = (\text{cont}^{\text{unit}} t (\bullet; \text{abort } t2\ 3))$ in the context of:

```
(% t2 (% t (% t2 ((k ()); event[α]) ...) ...) (λn. event[β]))
```

`k` is invoked nested inside multiple prompts of different tags: when `k` is restored, it will discard and replace the inner `t2` prompt (white background) entirely. This is important: after context restoration, the `abort` inside `k` will be executed, passing `3` to the *outermost* handler and emitting event β : the innermost `t2` prompt contains an abort to `t2`, yet the innermost handler will not execute; the *outermost* handler will.

```
(% t2 (% t ((); abort t2 3) ...) (λn. event[β])) ⇒* (λn. event[β]) 3
```

Our initial effect for invoking continuations with further control effects (Use Case 4) would have the inner prompt’s body effect contain `abort t2 {ε}`, which our suggested prompt handling would then join into the *innermost* prompt due to the matching tag. This has two problems: it spuriously suggests the innermost prompt’s handler could execute, introducing a kind of imprecision; and because that abort effect would no longer be present in the effect of the outermost prompt’s body, it would be missed that the outer handler *could* run, making the approach unsound. We must refine our approach for these “jumps.”

23:12 Lifting Sequential Effects to Control Operators

$c \in \text{ControlEffect} ::=$	$\text{replace } \ell : Q \rightsquigarrow \tau$ $ \text{abort } \ell Q \rightsquigarrow \tau$ $ \boxed{c}_\ell$	Invoked continuation up to ℓ , with prefix & underlying effect Q , continuation result type τ Abort up to ℓ after effects Q , throwing value of type τ Blocked control effect, frozen until nearest prompt for ℓ (triggered inside restored continuation targeting ℓ)
$p \in \text{Prophecy} ::=$	$\text{prophecy } \ell \chi \rightsquigarrow \tau \text{ obs } \chi'$ $ \boxed{p}_\ell$	Prophecy of latent continuation effect χ , effect χ' observed since point of prophecy (call/cc) Prophecy blocked until ℓ
$\chi \in \text{ContinuationEffect} =$	$\text{set Prophecy} \times \text{set ControlEffect} \times \text{option UnderlyingEffect}$	

■ **Figure 2** Grammar of continuation effects over an existing effect quantale $Q \in \text{UnderlyingEffect}$.

We resolve this by adding one final nuance to control effect sets. We allow control effects to be either basic (the **replace** or **abort** effects we have already seen, with target tags) or *blocked* \boxed{c}_ℓ for a control effect c . A blocked control effect \boxed{c}_ℓ is ignored by prompts (left in the control effect set) until a prompt tagged t is reached – that prompt will *unblock* the effect, leaving c in the control effect set of the prompt. When invoking a continuation, instead of simply including the latent control effects in the effect of the invocation, the type rule will include the latent control effects *blocked* until the target tag of the continuation. So in the example above, the inner prompt’s body effect would instead include $\boxed{\text{abort } t2 \{ \epsilon \}}_t$. This would be ignored (propagated) by the inner prompt’s typing (so the inner handler would not be spuriously considered), unblocked by the middle prompt’s typing, and finally resolved in the outer prompt’s typing (which would trigger consideration of the outer handler). Because the overall effect of any execution path that triggers such blocked control effects must still execute code along the way to the continuation invocation, blocked control effects still accumulate on the left.

Prophecies raise similar issues that lead to similar introduction of blocked prophecies: if \mathbf{k} above instead captured a continuation up to a prompt tagged $t2$, the white-background portion of the term discarded when \mathbf{k} was invoked would not be part of the captured continuation, so it should not be incorporated into the observation part of a prophecy. Because of this, blocked prophecies *must not accumulate while blocked*. The difference is this: blocked control effects continue to accumulate on the left because control operations do not discard code that occurs on the way to a control effect, while blocked prophecies must “skip over” the code discarded by control operations, which always appears to the right (later in source program order).

4 Continuation Effects

This section makes the outlines of the previous section precise, and fills in missing details (such as coordinating the type of a value thrown with the argument type of a handler). Figure 2 defines the effects of $\mathcal{C}(Q)$ derived from the examples above, for underlying effect quantale Q . Continuation-aware effects of an underlying effect quantale Q are effects χ of three components: a prophecy set P , a control effect set C , and an optional underlying effect \underline{Q} . Basic control effects include effects representing aborts to a tagged prompt (**abort** $\ell Q \rightsquigarrow \tau$) or invoking continuations that replace the context up the nearest tagged prompt (**replace** $\ell : Q \rightsquigarrow \tau$), as suggested by Use Cases 2 and 3. These versions are additionally tagged with the specific prompt tag they target (ℓ), and each carries a type τ – for replacement this is the result

type of the restored continuation, and for aborts this is the type of the value thrown (each intuitively a kind of result type for each control behavior). Both control effects and prophecies may also be blocked until a prompt with a certain tag if they originate inside a continuation that was invoked (per Use Case 6). Note that blocking constructors may nest arbitrarily deeply, because one restored continuation may restore another continuation which may restore another. . . and so on. For a term with effect (P, C, \underline{Q}) :

- The prophecy set P contains prophecies for all uses of `call/cc` within the term (some possibly blocked if introduced by restoring a continuation).
- The control effect set C describes all possible exits of the term via control operations (abort or continuation invocation). For aborts of continuation invocation that occur directly, C will contain basic control effects. For aborts or continuation invocation that may occur in the body of a restored continuation, there will be blocked control effects.
- The underlying effect \underline{Q} describes (an upper bound on) the underlying effect of any execution of the term that does not exit via control operator.

To define sequencing and join formally, we must first lift the underlying effect quantale's operators to deal with missing effects:

$$\underline{Q}_1 \triangleright \underline{Q}_2 = \begin{cases} \top & \text{if } \underline{Q}_1 = \top \vee \underline{Q}_2 = \top \\ \perp & \text{if } \underline{Q}_1 = \perp \vee \underline{Q}_2 = \perp \\ \underline{Q}_1 \triangleright \underline{Q}_2 & \text{otherwise} \end{cases} \quad \underline{Q}_1 \sqcup \underline{Q}_2 = \begin{cases} \top & \text{if } \underline{Q}_1 = \top \vee \underline{Q}_2 = \top \\ \underline{Q}_1 & \text{if } \underline{Q}_2 = \perp \\ \underline{Q}_2 & \text{if } \underline{Q}_1 = \perp \\ \underline{Q}_1 \sqcup \underline{Q}_2 & \text{otherwise} \end{cases}$$

We also require a way to prefix control effects with an underlying effect, to implement left accumulation (recursively), extending the ideas of Use Cases 2, 3, and 6 to the final definition:

$$\begin{aligned} Q \triangleright \boxed{c}_\ell &= \boxed{Q \triangleright c}_\ell \\ Q \triangleright \text{replace } \ell : Q' \rightsquigarrow \tau &= \text{replace } \ell : (Q \triangleright Q') \rightsquigarrow \tau \\ Q \triangleright \text{abort } \ell : Q' \rightsquigarrow \tau &= \text{abort } \ell : (Q \triangleright Q') \rightsquigarrow \tau \end{aligned}$$

and we will lift this to operate on control effect *sets* and possibly-absent underlying effects:

$$\underline{Q} \triangleright C = \text{if } (\underline{Q} = \perp) \text{ then } \emptyset \text{ else } (\text{map } (Q \triangleright _) C)$$

Likewise, we must define a means of right-accumulating in prophecies:

$$\begin{aligned} \overline{\text{prophecy } \ell (P, C, \underline{Q}) \rightsquigarrow \tau \text{ obs } (P', C', \underline{Q}')}_{\ell'} \blacktriangleright \chi'' &= \overline{\text{prophecy } \ell (P, C, \underline{Q}) \rightsquigarrow \tau \text{ obs } (P', C', \underline{Q}')}_{\ell'} \\ \overline{\text{prophecy } \ell (P, C, \underline{Q}) \rightsquigarrow \tau \text{ obs } (P', C', \underline{Q}') \blacktriangleright (P'', C'', \underline{Q}'')}_{\ell'} \\ &= \overline{\text{prophecy } \ell (P, C, \underline{Q}) \rightsquigarrow \tau \text{ obs } ((P' \blacktriangleright (P'', C'', \underline{Q}'')) \cup P'', C' \cup (\underline{Q}' \triangleright C''), \underline{Q}' \triangleright \underline{Q}'')}_{\ell'} \end{aligned}$$

which we also lift to operate on prophecy *sets* (not shown, but analogous to the lifting of left-accumulation). Finally, this is enough to define sequencing and join:

$$\begin{aligned} (P_1, C_1, \underline{Q}_1) \sqcup (P_2, C_2, \underline{Q}_2) &= (P_1 \cup P_2, C_1 \cup C_2, \underline{Q}_1 \sqcup \underline{Q}_2) \\ (P_1, C_1, \underline{Q}_1) \triangleright (P_2, C_2, \underline{Q}_2) &= ((P_1 \blacktriangleright (P_2, C_2, \underline{Q}_2)) \cup P_2, C_1 \cup (\underline{Q}_1 \triangleright C_2), \underline{Q}_1 \triangleright \underline{Q}_2) \end{aligned}$$

Joins are implemented component-wise, using set union on prophecy or control effect sets, and the (option-lifted) join from the underlying effect quantale. The sequencing operator, and its relation to the accumulation on prophecies, is a bit complex and warrants some further explanation. The sequence operator \triangleright is defined according to the ideas driven by Use Cases 2, 3, and 5. Underlying effects are sequenced by reusing the underlying effect quantale. Control effects are handled by left-accumulating. In fact, in the case where there are no prophecies (`call/ccs`) involved, the handling of the control effect sets and underlying effects is exactly as in Use Case 2, just extended for the additional control effects. Deferring

the right-accumulation \blacktriangleright for one further moment, the full sequencing operator produces a resulting prophecy set as the union of prophecies from the second effect (which are unaffected by the first) with the result of the first effect’s prophecies accumulating effects from the right, since that effect will (in the type rules) correspond to behavior that will be part of the captured continuation. The right accumulation for prophecies implemented by \blacktriangleright essentially just implements sequential composition of the observation with the accumulated effect – the recursive use of \blacktriangleright could equivalently just be \triangleright , but direct recursion is easier to prove things about than mutual recursion. As suggested by Use Case 6, blocked prophecies do not accumulate. It is easy to confirm that the unit element for \triangleright is $(\emptyset, \emptyset, I)$, where I is the identity of the underlying effect quantale.

\blacktriangleright Remark 2 (Sets of Underlying Effects). We have described most of the structure of lifting an effect quantale to support delimited control: sequencing and join operators that distribute over each other, along with a unit for sequencing. We have not yet stated whether the result $\mathcal{C}(Q)$ of the lifting is an effect quantale. As described so far, it is not quite an effect quantale: there is no single distinguished top in the partial order induced by \sqcup : for any effect (P, C, \underline{Q}) , a larger effect can be obtained by adding new control effects or prophecies. And because the underlying \top can appear in multiple ways, conceptually many different incomparable effects should be considered erroneous. Introducing a distinguished top element Err , and wrapping the sequencing and join definitions above with an additional operation producing Err any time the operations above produce effects containing underlying top. This would produce an effect quantale, but we take an alternative approach that also enhances flexibility, without adding special cases for \top throughout the system.

Using *sets* of structures containing underlying effects can lead to the extra set structure being “too picky” in distinguishing effects, in the sense of distinguishing intuitively equivalent effects. Consider the following join:

$$(\emptyset, \{\text{abort } \ell \ \{\alpha\}\}, \perp) \sqcup (\emptyset, \{\text{abort } \ell \ \{\beta\}\}, \perp) = (\emptyset, \{\text{abort } \ell \ \{\alpha\}, \text{abort } \ell \ \{\beta\}\}, \perp)$$

which could be the effect of `if c (event[α]; abort ℓ 3) (event[β]; abort ℓ 3)`. Their join is *not* the very similar $(\emptyset, \{\text{abort } \ell \ \{\alpha, \beta\}\})$, which also indicates an abort after one of the two same underlying effects, and is the effect of a minor rewrite of the last expression: `(if c (event[α]) (event[β])) ; abort ℓ 3`. Both effects indicate executing α or β before aborting, and replacing one with the other inside a prompt body will not change the *prompt*’s effect (per Use Cases 2 and 3): the prompt will sequence each with the handler effect, and join them together). Yet because \sqcup is defined using set union, they are incomparable in the induced partial order $x \sqsubseteq y \leftrightarrow x \sqcup y = y$, because joining them yields a *third* effect: $(\emptyset, \{\text{abort } \ell \ \{\alpha\}, \text{abort } \ell \ \{\beta\}, \text{abort } \ell \ \{\alpha, \beta\}\}, \perp)$. This is not a valuable distinction. We would like at least $(\emptyset, \{\text{abort } \ell \ \{\alpha\}, \text{abort } \ell \ \{\beta\}\}, \perp) \sqsubseteq (\emptyset, \{\text{abort } \ell \ \{\alpha, \beta\}\})$ because every abort prefix on the left is over-approximated by an abort prefix on the right.² One way to achieve this would be to replace the set union of control effect sets or prophecy sets in the sequencing and join operators with operators that also recursively joined the underlying effects of all aborts to the same tag, joined the underlying effects of all replacements to the same tag, and joined the observed effects of all prophecies to the same tag with the same prediction (plus joining types). The partial order induced by this modification would establish this desirable order.

Directly extending \sqcup and \triangleright as given to *both* recursively join when combining sets and lift underlying \top to a special Err would yield a proper effect quantale, but add significant complexity that is orthogonal to the key ideas of our approach. *Instead*, we make two

² While in this $\mathcal{T}(\Sigma)$ example the effects are equivalent, examples in other effect quantales only justify \sqsubseteq .

$$\begin{aligned}
C_1 \sqsubseteq C_2 &= \bigwedge \left\{ \begin{array}{l} \forall \ell, Q, \tau. \text{replace } \ell : Q \rightsquigarrow \tau \in C_1 \Rightarrow \exists Q'. \text{replace } \ell : Q' \rightsquigarrow \tau \in C_2 \wedge Q \sqsubseteq Q' \\ \forall \ell, Q, \tau. \text{abort } \ell Q \rightsquigarrow \tau \in C_1 \Rightarrow \exists Q'. \text{abort } \ell Q' \rightsquigarrow \tau \in C_2 \wedge Q \sqsubseteq Q' \end{array} \right. \\
P_1 \sqsubseteq P_2 &= \\
(\forall \ell, \chi, \chi', \tau. \text{prophecy } \ell \chi \rightsquigarrow \tau \text{ obs } \chi' \in P_1 \Rightarrow \exists \chi''. \text{prophecy } \ell \chi \rightsquigarrow \tau \text{ obs } \chi'' \in P_2 \wedge \chi' \sqsubseteq \chi'') \wedge \\
(\forall \ell, \ell', \chi, \chi', \tau. \boxed{\text{prophecy } \ell \chi \rightsquigarrow \tau \text{ obs } \chi'}_{\ell'} \in P_1 \Rightarrow \exists \chi''. \boxed{\text{prophecy } \ell \chi \rightsquigarrow \tau \text{ obs } \chi''}_{\ell'} \in P_2 \wedge \chi' \sqsubseteq \chi'') \\
(P, C, Q) \sqsubseteq (P', C', Q') &\leftrightarrow P \sqsubseteq P' \wedge C \sqsubseteq C' \wedge Q \sqsubseteq Q' \\
\chi \approx \chi' &= (\text{NoUnderlyingTop}(\chi, \chi') \wedge \chi \sqsubseteq \chi' \wedge \chi' \sqsubseteq \chi) \vee (\text{UnderlyingTop}(\chi) \wedge \text{UnderlyingTop}(\chi'))
\end{aligned}$$

■ **Figure 3** Direct partial order and equivalence on continuation effects.

adjustments. First, we define a direct partial order on continuation effects in Figure 3, where an effect χ is less than another effect χ' if every prophecy observation, abort effect, replacement effect, or underlying effect in χ is over-approximated by *some* corresponding component in χ' . This captures the intuition behind the desirable ordering outlined above. We also define a corresponding equivalence relation \approx on continuation effects, which equates all continuation effects containing an underlying \top (anywhere in the effect) and otherwise uses the partial order to induce equivalence. In the type rules considered in Section 4, this is the notion of subeffecting used (rather than the traditional partial order derived from a join), and all effects are considered modulo the equivalence relation. Quotienting $\mathcal{C}(Q)$ by the equivalence \approx yields a proper effect quantale, equivalent to the direct but verbose version outlined above. See our technical report [32]) for further details.

We consider a type-and-effect system for a language with the constructs from Figure 1. Our extended technical report [32] extends these results for composable continuations. Our expressions and types are:

$$\begin{aligned}
\text{Expressions } e &::= p \mid \lambda x. e \mid (e e) \mid (\% \ell E v) \mid (\text{call/cc } \ell e) \mid (\text{abort } \ell e) \mid (\text{cont } \ell E) \mid \text{if } e e e \\
\text{Values } v &::= (\lambda x. e) \mid \text{cont } \ell E \mid v_p \\
\text{Types } \tau, \gamma &::= \text{unit} \mid \text{bool} \mid \tau \xrightarrow{\chi} \tau \mid (\text{cont } \ell \tau \chi \tau) \mid \mu X. \tau
\end{aligned}$$

p and v_p are parameters of the system following Gordon’s work [30, 31]: primitives (which can include operations such as locking primitives or `event`[α]) and primitive values (e.g., for encoding locations). Gordon’s soundness framework also parameterizes operational semantics by an abstract notion of state, and semantics for primitives manipulating state; we assume (and later use) a similar framework, which admits a range of concrete examples.

Types include common primitive types, function types with latent effects, equirecursive types (needed for typing loops), as well as a type for continuation values that we discuss with the type rule for invoking continuations. The type rules for lambda abstraction, function application, and conditionals are as in Section 2 (though using continuation effects), so we do not discuss them further. Typing uses of primitives requires additional rules and parameters to define the additional types (e.g., lock or location types) and their relationship to operational primitives, following Gordon [30, 31]. For intuition, readers may assume p is simply the `event`[$_$] primitive from our running example. We include subtyping ($<:$), including the standard type-and-effect subsumption rule, and function subtyping that is covariant in the body’s latent effect. Figure 4 gives central type rules for this paper, for prompts, aborts, continuation capture, and continuation invocation.

We will discuss the type rules in relation to the Use Cases from Section 3.

T-ABORT handles Use Case 2. As suggested in that discussion, the effect of an abort is to introduce a control effect signifying an abort to the targeted label, with an absent underlying effect. The initial prefix of the abort is “empty” – the underlying unit effect I – and the

23:16 Lifting Sequential Effects to Control Operators

$$\begin{array}{c}
\forall \tau', Q. (\text{abort } \ell \ Q \rightsquigarrow \tau') \in \overline{[C]_\ell} \Rightarrow \tau' <: \sigma \quad \forall Q, \tau'. (\text{replace } \ell : Q \rightsquigarrow \tau') \in \overline{[C]_\ell} \Rightarrow \tau' <: \tau \\
\left(\begin{array}{c} \forall \chi_{\text{prophecy}}, \tau', P_p, C_p, Q_p. \text{prophecy } \ell \ \chi_{\text{prophecy}} \rightsquigarrow \tau' \text{ obs } (P_p, C_p, Q_p) \in \overline{[P]_\ell} \Rightarrow \\ \overline{[P]_\ell} \sqsubseteq \overline{[P_{\text{prophecy}}]_\ell} \wedge \overline{[C]_\ell} \sqsubseteq \overline{[C_{\text{prophecy}}]_\ell} \wedge Q_p \sqsubseteq Q_{\text{prophecy}} \wedge \tau <: \tau' \end{array} \right) \\
\text{V-EFFECTS} \frac{}{\text{validEffects}(P, C, Q, \ell, \tau, \sigma)} \\
\\
\text{T-PROMPT} \frac{\Gamma \vdash e : \tau \mid (P, C, Q) \quad \Gamma \vdash h : \sigma \xrightarrow{(\emptyset, \emptyset, Q_h)} \tau \mid I \quad \text{validEffects}(P, C, Q, \ell, \tau, \sigma)}{\Gamma \vdash (\% \ell \ e \ h) : \tau \mid (\overline{[P]_\ell} \setminus Q_h \ \ell, \overline{[C]_\ell} \setminus \ell, \underline{Q} \sqcup (\bigsqcup \overline{[C]_\ell}^{Q_h}))} \\
\\
\text{T-CALLCONT} \frac{\text{NonTrivial}(\chi_k) \quad \Gamma \vdash e : (\text{cont } \ell \ \tau \ \chi_k \ \gamma) \xrightarrow{\chi} \tau \mid \chi_e}{\Gamma \vdash (\text{call/cc } \ell \ e) : \tau \mid (\chi_e \triangleright \chi) \triangleright (\{\text{prophecy } \ell \ \chi_k \rightsquigarrow \gamma \text{ obs } (\emptyset, \emptyset, I)\}, \emptyset, I)} \\
\\
\text{T-APPCONT} \frac{\Gamma \vdash k : \text{cont } \ell \ \tau' (P, C, Q) \ \tau'' \mid \chi_k \quad \Gamma \vdash e : \tau' \mid \chi_e}{\Gamma \vdash (k \ e) : \tau \mid \chi_k \triangleright \chi_e \triangleright (\overline{[P]_\ell}, \overline{[C]_\ell} \cup (\underline{Q} \triangleright \{\text{replace } \ell : I \rightsquigarrow \tau''\}), \perp)} \\
\\
\text{T-ABORT} \frac{\Gamma \vdash e : \tau \mid \chi_e}{\Gamma \vdash \text{abort } \ell \ e : \sigma \mid \chi_e \triangleright (\emptyset, \{\text{abort } \ell \ I \rightsquigarrow \tau\}, \perp)}
\end{array}$$

■ **Figure 4** Typing control operators with continuation effects.

control effect tracks the type of the thrown value. The overall effect of an **abort** expression sequences this after the effect of reducing e to a value, since this occurs earlier in evaluation order than the abort operation itself. A subtlety worth noting, is that this also ensures a **call/cc** inside e will correctly accumulate the pending abort in the captured context.

T-APPCONT handles Use Cases 3 and 4. First consider the basic case where the invoked continuation has only simple underlying effects (i.e., P and C in the continuation's latent effect are both \emptyset). As with T-ABORT, subterms are reduced χ_k to values before the control behavior occurs, so those effects are sequenced before the control behavior itself. With that taken care of, we may temporarily assume both k and e are already syntactic values to simplify discussion of effect. In this case the rule simplifies to exactly what the example in Use Case 3 suggested: the underlying effect is absent (because the term does not return normally, but via a control behavior), and a control effect is introduced reflecting that a continuation with underlying effect \underline{Q} is invoked. A subtlety here is that we cannot simply write $\text{replace } \ell : \underline{Q} \rightsquigarrow \tau''$, because \underline{Q} may be \perp , which would make that control effect invalid. Instead we (ab)use the left-accumulation operator on control effects: prefixing the *unit-effect* replacement effect with \underline{Q} will give the expected result when \underline{Q} is present, and otherwise result in the empty set. The replacement effect also records the result type of the invoked continuation, and the rule also ensures the argument provided to the continuation is the expected type.

T-APPCONT also handles Use Case 4, adjusted per Use Case 6: the latent control effects of the continuation are included, but *blocked until* ℓ , to ensure no prompt rule (discussed shortly) resolves those behaviors before the behaviors escape a prompt tagged ℓ . Unlike the discussion in Section 3, this finished rule also supports the case where the restored continuation contains (possibly-nested) uses of **call/cc**, blocking the latent prophecies as well.

The conclusion of T-APPCONT critically overloads the syntax for constructing blocked prophecies and control effects, to block prophecy *sets* and control effect *sets*. This overload *almost* maps the appropriate blocking constructor over each set – however, it first checks that this will not result in a control effect of the form $\overline{[C]_\ell}$ for some tag ℓ (similarly for prophecies). Such a control effect would represent a control effect that should propagate directly through *two* prompts tagged ℓ , but this is dynamically impossible: any number of nested restorations of continuations to the same tag remains within the same prompt, e.g.:

```
(% ℓ ((cont ℓ (•; ((cont ℓ (•; abort ℓ 5)) 4))) 3) h)
⇒ (% ℓ (3; ((cont ℓ (•; abort ℓ 5)) 4)) h) ⇒ (% ℓ ((cont ℓ (•; abort ℓ 5)) 4) h)
⇒ (% ℓ (4; abort ℓ 5) h) ⇒ (% ℓ (abort ℓ 5) h) ⇒ (h 5)
```

Naïvely mapping the blocking constructor would yield the control effect set $\{\llbracket \text{abort } \ell I \rightsquigarrow \text{nat} \rrbracket_{\ell}\}$ for the body; our discussion with Use Case 6 about each prompt removing one layer of blocking (which we will see does inform T-PROMPT) would then not pair the abort with the local handler that is invoked. With the modified mapping, the simplification of the control effect set is instead $\{\llbracket \text{abort } \ell I \rightsquigarrow \text{nat} \rrbracket_{\ell}\}$ (only one layer of blocking), which will match the `abort` to the correct handler.

T-CALLCONT is a bit more subtle. Standard for any type rule for `call/cc`, the rule ensures the result type of the expression itself (τ) is also the return type of the body function (for executions that return normally) and the argument type assumed for the continuation (since the location of the `call/cc` becomes the hole the argument replaces at invocation). As suggested in the discussion of Use Case 5, the effect arising from the use of `call/cc` itself is a prophecy effect, recording the assumed latent effect of the captured continuation and the assumed result type of the captured continuation – both of which make their way into the assumed type of the argument to the `call/cc` body. The initial observation is empty, because this effect corresponds intuitively to the point in the execution from which the prediction begins – but this point is the heart of another key subtlety. As with other rules, the subterm e must be reduced before anything else, so its effect is sequenced before others. But naïvely ordering the body’s (latent) effect would place it after the prophecy, as dynamically the continuation is captured before the body is evaluated (since the continuation becomes the argument in E-CALLCC). However, this would result in the prophecy effect observing the body of the `call/cc` – which is incorrect, as that behavior will not be part of the captured continuation. Thus we place the prophecy *after* the body’s latent effect.

For a small variation on Use Case 5’s first example, this gives us:

$$\begin{array}{c} \chi_e \triangleright (\{\text{prophecy } t (\emptyset, \emptyset, \{\alpha\} \text{ obs } (\emptyset, \emptyset, I)), \emptyset, I\} \quad (\emptyset, \emptyset, \{\alpha\}) \\ (\% t (\underbrace{\text{call/cc } t (\lambda k. e)}_{\chi_e \triangleright (\{\text{prophecy } t (\emptyset, \emptyset, \{\alpha\} \text{ obs } (\emptyset, \emptyset, \{\alpha\})), \emptyset, \{\alpha\})} \quad ; \text{event}[\alpha]) \dots) \end{array}$$

The individual effects of the `call/cc` and `event` expressions (given above the term) simplify to the effect below the term. The prophecy from the `call/cc` observes the event that would be captured in its context. Because e ’s effect χ_e is to the *left* of the resulting prophecy, e ’s non-captured behavior is *not* observed, and the prophecy under the term will appear in the prophecy set of the overall prompt’s body.

T-CALLCONT has one extra antecedent, constraining the prophecy to predict a *non-trivial* effect, to avoid degeneracy. The simplest problematic prediction would be to predict an effect of $(\emptyset, \emptyset, \perp)$. Consider the effect of the code `event`[α]; (`k` `()`). The effect would be $(\emptyset, \emptyset, \{\alpha\}) \triangleright (\emptyset, \emptyset, \perp) = (\emptyset, \emptyset, \perp)$. This is problematic: the term *does* have a behavior, but the effect reflects *no* behavior. This could be introduced by a circularity:

```
(% t (let k = (call/cc t (λk. k)) in (event[α]; (k ()))) ...)
```

This term is in fact the macro-expansion for an infinite loop that executes the event forever. Assuming the degenerate latent effect in the `call/cc` gives `k` the degenerate effect, which gives the body of the the let-expression – the context captured by `call/cc` – a degenerate effect, allowing the observed effect to match the prophecy (both degenerate). Requiring a non-empty control effect set or underlying effect avoids this collapse. (This is not a termination-sensitivity issue; a terminating while loop has the same challenge, but a larger term.)

$$\begin{aligned}
 P \setminus^Q \ell &= \{p \setminus^Q \ell \mid p \in P \wedge \text{OuterTag}(p) \neq \ell\} & C \setminus \ell &= \{c \in C \mid \text{OuterTag}(c) \neq \ell\} \\
 C|_{\ell}^Q &= \{Q' \triangleright Q \mid \text{abort } \ell \ Q' \rightsquigarrow _ \in C\} \cup \{Q' \mid \text{replace } \ell : Q' \rightsquigarrow _ \in C\} \\
 \overline{\text{prophecy } \ell' (P, C, Q) \rightsquigarrow \tau \text{ obs } (P', C', Q')}_{\ell} &= \text{prophecy } \ell' (P, C, Q) \rightsquigarrow \tau \text{ obs } (\overline{P}_{\setminus \ell}^{\ell'}, \overline{C}_{\setminus \ell}^{\ell'}, Q') \\
 \overline{\text{prophecy } \ell' (P, C, Q) \rightsquigarrow \tau \text{ obs } (P', C', Q')}_{\ell} &= \text{prophecy } \ell' (P, C, Q) \rightsquigarrow \tau \text{ obs } (\overline{P}_{\setminus \ell}^{\ell'}, \overline{C}_{\setminus \ell}^{\ell'}, Q') \\
 \overline{\text{prophecy } \ell' (P, C, Q) \rightsquigarrow \tau \text{ obs } (P', C', Q')}_{\ell''} &= \overline{\text{prophecy } \ell' (P, C, Q) \rightsquigarrow \tau \text{ obs } (P', C', Q')}_{\ell''} \quad (\text{if } \ell \neq \ell'') \\
 (\text{prophecy } \ell' \chi \rightsquigarrow \tau \text{ obs } (P', C', Q')) \setminus^Q \ell &= \text{prophecy } \ell' \chi \rightsquigarrow \tau \text{ obs } (P' \setminus^Q \ell, C' \setminus \ell, Q' \sqcup (C'|_{\ell}^Q)) \\
 (\overline{\text{prophecy } \ell' \chi \rightsquigarrow \tau \text{ obs } (P', C', Q')}_{\ell}) \setminus^Q \ell &= \text{prophecy } \ell' \chi \rightsquigarrow \tau \text{ obs } (P' \setminus^Q \ell, C' \setminus \ell, Q' \sqcup (C'|_{\ell}^Q)) \\
 (\text{prophecy } \ell' \chi \rightsquigarrow \tau \text{ obs } (P', C', Q') \text{ until } \ell'') \setminus^Q \ell &= \text{prophecy } \ell' \chi \rightsquigarrow \tau \text{ obs } (P', C', Q') \text{ until } \ell'' \quad (\text{if } \ell \neq \ell'')
 \end{aligned}$$

■ **Figure 5** Auxilliary definitions used by type rules.

T-PROMPT is the most complex type rule in the system, because it serves many roles. In addition to giving an overall type and effect to the prompt, it must check that:

- Any **abort** to this handler throws values whose type is a subtype of the handler’s argument.
- Any continuation invoked in the body that will restore up to this prompt has a result type that is a subtype of the prompt’s own result type.
- Any **call/cc** that captures a continuation delimited by this prompt was typed assuming the result was a *supertype* of the prompt’s actual result type.
- Any **call/cc** that captures a continuation delimited by this prompt was typed assuming a valid latent effect for that continuation (i.e., that prophecies targeting that tag are upper-bounds on the observations).

The first three checks are handled by the subtyping constraints in the auxiliary judgment V-EFFECTS. Notice that the antecedents of V-EFFECTS quantify over elements of *unblocked* control effect sets and prophecy sets. Unblocking, written $\overline{_}_{\ell}$ is defined for prophecies in Figure 5: it maps a corresponding per-element unblocking operation, which is a no-op on elements that were not blocked, a no-op on elements blocked until a *different* tag, and strips off a block constructor for ℓ if that is the outermost constructor. Unblocking for control effect sets is defined analogously. Intuitively, this corresponds to the fact that any control effect or prophecy blocked until a prompt for ℓ has now *reached* a prompt for ℓ .

The last of the checks above is handled by the prophecy-related antecedent of V-EFFECTS, which *nearly* checks that the observations for a given prophecy are a subeffect of what was predicted – that would be $(P_p, C_p, Q_p) \sqsubseteq \chi_{\text{proph}}$, which would be sound but overly conservative. Instead, the comparison of the prophecy and observation, compares the *unblocked* versions of prophecy and control effect sets: $\overline{P}_{\setminus \ell}^{\ell} \sqsubseteq \overline{P}_{\text{proph}}^{\ell}$ and $\overline{C}_{\setminus \ell}^{\ell} \sqsubseteq \overline{P}_{\text{proph}}^{\ell}$. This is useful because if a context captured by **call/cc** contains an invocation of itself, the prophecy arising from T-CALLCONT will observe a *blocked* version of *its own prophecy* (from T-APPCONT), making the naive subeffect check too conservative: no prophecy can predict a blocked version of itself. Rather than being an esoteric concern, this is actually quite practical: encodings of loops using delimited continuations do exactly this. Unblocking for the prompt’s tag in V-EFFECTS resolves this: just like the quantifications unblock because those sets have now reached the corresponding prompt, the prophecy validation must reflect that the observation has now “reached” the corresponding prompt.

Let us revisit the example of Use Case 5 discussed above with T-CALLCONT. The prophecy in that derivation contains no blocked prophecies, so T-PROMPT will effectively check that the observation is less than the prophecy – that $(\emptyset, \emptyset, \{\alpha\}) \sqsubseteq (\emptyset, \emptyset, \{\alpha\})$, which trivially succeeds because the prophecy predicted its context’s behavior exactly.

We can see the role of unblocking more clearly by revisiting the motivating example for requiring prophecized effects to be non-trivial (eliding types for brevity).

$$(\% t \text{ (let } k = \underbrace{(\text{call/cc } t \text{ } (\lambda k. k))}_{(\{\text{prophecy } t \text{ } (\emptyset, \{\text{replace } t \text{ } \alpha^*\}, \perp) \text{ obs } (\emptyset, \emptyset, I))\}, \emptyset, I)} \text{ in } \underbrace{(\text{event}[\alpha]; (k \text{ } ()))}_{(\emptyset, \emptyset, \{\alpha\}) \text{ } (\emptyset, \{\text{replace } t \text{ } \alpha^*\}, \perp)})) \dots)$$

The effect of the prompt's body is the sequencing (with \triangleright) of the three individual subterm effects written above the term fragments, writing α^* for the set of all finite traces consisting of only α . Simplifying the sequencing of the two right-most effects first will result in a control effect set $\{\text{replace } t \text{ } (\{\alpha\} \triangleright \alpha^*)\}_t$ (invoke the continuation after an α event, with aggregate behavior of some non-zero finite number of α events). Simplifying again, the prophecy will observe this, and T-PROMPT will check (via V-EFFECTS) that $\{\text{replace } t \text{ } (\{\alpha\} \triangleright \alpha^*)\}_t^1 \sqsubseteq \{\text{replace } t \text{ } \alpha^*\}_t$, which is equivalent to checking $\{\text{replace } t \text{ } (\{\alpha\} \triangleright \alpha^*)\} \sqsubseteq \{\text{replace } t \text{ } \alpha^*\}$, which is true because $\{\alpha\} \triangleright \alpha^* \sqsubseteq \alpha^*$ in $\mathcal{T}(\Sigma)$ (the set of non-empty finite traces containing only α is a subset of the set of possibly-empty finite traces containing only α).

Finally, T-PROMPT must give a type and effect to the prompt expression itself. The underlying effect must include (1) the body's underlying effect, (2) the effect of any possible paths through the body that abort to the local handler and execute it (including those resulting from invoking continuations up to this prompt prior to aborting), and (3) the effect of any possible paths through the body that invoke one or more continuations up to this prompt before returning normally. (1) is simply Q . (2) is the result of sequencing any abort prefix for ℓ in the (unblocked) control effect set (which will incorporate aborts resulting from continuation invocation as well). (3) is simply the set of replacement effects in the (unblocked) C . (2) and (3) are computed via a projection operator $-|_{\ell}^{Q_h}$ applied to the unblocked control effect set, defined in Figure 5. The superscript on the operator is the underlying effect of the handler (constrained to have no control effects), and the subscript is the choice of relevant prompt tag. The resulting set of underlying effects is joined together with the body's underlying effect.

The prophecy and control effect sets of the overall prompt should propagate prophecies and control effects targeting other prompts, and remove those targeting the prompt at hand. To this end, the conclusion of T-PROMPT unblocks both sets and then removes (1) prophecies related to this prompt (which have now been validated) and (2) control effects related to this prompt (which have been incorporated into the prompt's underlying effect, because they address control behaviors scoped to this prompt). The (unblocked) control effects are simply filtered with $-\setminus \ell$ (Figure 5), which retains only basic control effects targeting other tags and control effects still blocked until other tags. (Thus, the nested control effect from Use Case 6 appears blocked in the effect of the innermost prompt.) The prophecy set is filtered similarly, but the filtered results are also transformed – the remaining observations must be adapted to model the changes to effects from going “through” this prompt. Thus the filtering operation $-\setminus^Q \ell$ on prophecy sets (Figure 5) selects those prophecies related to other prompts, then recursively transforms their observations – recursively filtering (and transforming) the observed prophecy and control effect sets, and joining transformations of their content into the observations' underlying effect, just as in the conclusion of T-PROMPT itself. This is important due to interactions between tags: a prophecy to an outer prompt may observe in its context aborts to an inner prompt: $-\setminus^Q \ell$ joins such abort prefixes with the handler that would run, and joins that into the underlying effect of the prophecy.

5 Iterating Continuation Effects

Prior work on effect quantales [30, 31] introduced the notion of lax iterability to introduce a loop iteration operator, as outlined in Section 2. We would like to reuse this operator construction for two reasons. First, we would like to check that if we macro-express loop constructs and derive rules for them as we proposed earlier, that they are consistent with manually-derived rules from prior work, which use the iteration operator. Second, the iteration operator has properties that make it useful for solving recursive constraints over effects, such as those that arise in building derived rules for control flow constructs and control operators later in the paper. Of course, lax iterability and the construction are defined on standard effect quantales,³ so do not apply directly to $\mathcal{C}(Q)$. Fortunately, one can apply the construction to $\mathcal{C}(Q)$ quotiented by \approx (a standard effect quantale), and since those elements are equivalence classes, simply use the behavior on elements to iterate in $\mathcal{C}(Q)$:

► **Theorem 3 (Lax Iterability with Continuations).** *For a laxly iterable underlying effect quantale Q , the effect quantale $\mathcal{C}(Q)/\approx$ is also laxly iterable, with the closure operator given by lifting the following operator from elements of $\mathcal{C}(Q)$ to the corresponding equivalence class.*

$$(P, C, \underline{Q})^* = \left(\bigcup_{i \in \mathbb{N}} P \blacktriangleright (P, C, \underline{Q})^i, \underline{Q}^* \triangleright C, \underline{Q}^* \right)$$

Notice that when $P = \emptyset$ and $C = \emptyset$, this specializes to the intuitive embedding of the (\perp -extended) underlying iteration: $(\emptyset, \emptyset, \underline{Q}^*)$. When only $P = \emptyset$ this specializes to $(\emptyset, \underline{Q}^* \triangleright C, \underline{Q}^*)$, intuitively reflecting that any control exits occur after repeating \underline{Q} some number of times first. While these examples “merely” drop certain components of Theorem 3, it helps to work from the simplest case up to the more complex versions, since the examples above correspond intuitively to various execution paths. The infinite union in the prophecy set is the most subtle part of the operation to explain. Consider an expression with the structure `while c (... (call/cc t ...) ...)`: Assume the tag t for the continuation captured inside the loop does not occur elsewhere inside the loop – in particular, that the captured continuation would extend *outside* the loop. Considering the runtime execution, in some sense the prophecy captured by the *first* loop iteration must predict not only the regular execution and exceptional executions of future iterations, but even the need for more prophecies to be generated by the `call/ccs` in future iterations as well! This is why the set of prophecies must still be sequenced with some form of themselves, rather than just some subset. During static typechecking, we must therefore conservatively overapproximate the number of iterations following a prophecy. It may be 0, 1, 2, ... or any number. So the approximation must consider all of those possibilities, hence the infinite union of finite repetitions following the prophecies. This requires prophecy sets to be possibly-infinite, but only countably so.

6 Type Safety

We have proven syntactic type soundness for the type system presented in Section 4. We summarize the proof here; see the technical report for full details [32]. We continue to reuse Gordon’s parameterization for soundness [30, 31], making the proof generic over a choice of abstract states (ranged over by σ) and related parameters subject to some restrictions.

³ Lax iterability is defined for a version of effect quantales using partial operators [31] instead of the distinguished \top used here to simulate partiality. But the definitions simplify to those used here when given total operations and using side-conditions to reject effects containing \top .

Progress is uninteresting (if primitives satisfy progress), in the sense that effects play no essential role (they are merely “pushed around” and the proof looks otherwise like standard progress proofs). Preservation is similar to the common formulation for syntactic type soundness results of sequential effect systems [34, 30, 68, 67]. It follows from single-step preservation: informally, for a well-typed runtime state σ and term e , if $\sigma; e \xrightarrow{q} \sigma'; e'$, then σ' and e' are also well-typed, and moreover if the static effects of e and e' are χ and χ' respectively, then $(\emptyset, \emptyset, q) \triangleright \chi' \sqsubseteq \chi$ – that is, sequencing the actual effect of the reduction with the residual effect of the reduced expression is soundly bounded by the effect of the original expression. This is the typical form of syntactic type safety proofs for sequential effect systems [34, 25].

Explaining the formal statement requires explaining the full details of parameterization. For brevity and lack of space, we offer the formal statement of single-step preservation specialized for our running example of $\mathcal{T}(\Sigma)$ and $\text{event}[-]$ with trivial (i.e., unit) state⁴ for which many of the conditions simplify to True:

► **Corollary 4** (Single-Step Preservation for $\mathcal{T}(\Sigma)$). *If $\Gamma \vdash e : \tau \mid \chi$ and $(\); e \xrightarrow{q} (\); e'$, then there exists a $\tau' <: \tau$ and a χ' such that $\Gamma \vdash e' : \tau' \mid \chi'$ and $q \triangleright \chi' \sqsubseteq \chi$.*

A key lemma for soundness of the rule for `call/cc` precisely relates prophecy observations to typing continuations. Informally, we prove that for a term $E[e]$, if the effect of e contains a prophecy $\text{prophecy } \ell \chi \rightsquigarrow \tau; \text{obs } (\emptyset, \emptyset, I)$ and E contains no prompts for tag ℓ , then (1) the effect of $E[e]$ contains a prophecy $\text{prophecy } \ell \chi \rightsquigarrow \tau \text{ obs } \chi'$ (accumulating some observation), and (2) plugging an appropriately-typed value v into E produces a term $E[v]$ with static effect χ' (exactly matching the observation). The assumptions of the lemma are exactly the conditions when considering E-CALLCC in the preservation proof: $E[e]$ is the body of the delimiting prompt and e is a use of `call/cc`, so by T-CALLCONT an appropriate prophecy exists in e 's effect ensuring the prophecy is given sound latent effect for typing the captured continuation.

► **Remark 5** (Syntactic vs. Semantic Soundness). Our proof imparts no semantic meaning to effects beyond syntactically relating the dynamic and static effects – it does not check that a certain effect enforces what it is meant to (e.g., deadlock freedom), unless like some finite trace effects [68, 45] the relation between static and dynamic effects is already the intent. This is common to any syntactic type safety approach for generic effects [51]. Gordon [31] has extended his proof approach for semantic pre- and post-condition type properties, such as ensuring locking effects accurately describe lock acquisition and release, but limited to safety properties; in principle this should be adaptable to our setting. Denotational approaches to abstract effect systems [42, 55, 70, 6] inherently give actual semantics, and therefore can ensure liveness properties.

7 Deriving Sequential Effect Rules

Section 4 developed the core type rules which give sequential effects to programs making direct use of tagged delimited control. As we have discussed, most programs do not use the full power of delimited control, and instead use only control flow constructs or weaker control operators. This section uses the new type type-and-effect rules to *derive* sequential effect rules for a range of control flow constructs and weaker control operators macro-expressed in terms of prompts.

⁴ Meaningful loops obviously require more meaningful state.

Our examples fall into two groups. First, we consider checking consistency of derived rules for typical control flow constructs with those hand-designed in prior work – for infinite loops and while loops. Second, we consider derived rules for constructs *never before addressed in sequential effect systems*: exceptions and generators.

In each case, we give a derived type rule for the construct of interest. While we are most explicit in Section 7.1, in each case our process for deriving the rule is the following:

1. Assume closed typing derivations for subexpressions (e.g., loop bodies)
2. Apply the type rules from Section 4 to give a closed-form rule for the macro’s expansion to be well-typed under the assumed subexpression types. Typically these have several undetermined choices for metavariables representing effects, with non-trivial constraints to close the typing derivation.
3. Simplify the type rule by giving solutions to the constrained-but-undetermined metavariables in terms of the subexpressions’ effects. This gives type rules that are simpler, and possibly less general, but given entirely in terms of the subexpressions’ effects. The simplifications are typically involve rewriting by the laws satisfied by $\mathcal{C}(Q)$, and using the iteration operator from prior work [30, 31] to solve recursive constraints on undetermined effects.

Due to space constraints, we show detailed derivations only for simple infinite loops (Section 7.1), giving only final results for others. The calculations for the remaining examples are done in the same way, and are available in the accompanying technical report [32].

7.1 Infinite Loops

Consider a simple definition of an infinite loop using the constructs we have derived here:

$$\llbracket \text{loop } e \rrbracket = (\% \ell \text{ (let } cc = (\text{call/cc } \ell \text{ } (\lambda k. k)) \text{ in } (\llbracket e \rrbracket; cc \text{ } cc)) \text{ } (\lambda _ . \text{tt}))$$

The term above executes e repeatedly, forever (assuming e does not abort). Thus, its effect *ought* to indicate that e ’s effect, which we take to be $(\emptyset, \emptyset, Q_e)$,⁵ is repeated arbitrarily many times. We take this expansion as the body of a macro $\llbracket \text{loop } e \rrbracket$. This program can be well-typed in our system, with an appropriate effect (assuming the underlying effect quantale is *laxly iterable* per Section 2). The body of the `call/cc` is pure, but for the expression to be well-typed, the `call/cc`’s own effect must prophesize some effect (\emptyset, C_p, Q_p) of the enclosing continuation up to the prompt for ℓ (because no `call/cc` occurs in the continuation of another, the prophecy set can be empty).

The right-accumulator of the prophecy effect, initially $(\emptyset, \emptyset, I)$, eventually accumulates a control set $(Q_e \triangleright \boxed{C_p}_\ell) \cup ((Q_e \triangleright Q_p) \triangleright \{\text{replace } \ell : I \rightsquigarrow \text{unit}\})$ and underlying effect \perp , because between capturing the continuation and the prompt, the program evaluates e (underlying effect Q_e) and then invokes the captured continuation (prophecized effect (\emptyset, C_p, Q_p) , underlying effect \perp). This is also the resulting control effect set for the body; we will refer to it as C . The type rule for the prompt itself removes all ℓ -related prophecies and control effects, leaving both empty (since we assume no control effects escape e , C_p should only contain ℓ -related effects, while the prophecy set contains the single prophecy from the `call/cc`). For the underlying effect, T-PROMPT joins the immediate underlying effect Q_e (from the overall judgment, not the prophecy) with all ℓ -related behaviors in C – e has no escaping control effects, and the macro-expanded loop contains no aborts, so C_p ought to have only replace effects, meaning C contains only replace effects, and $Q_e \sqcup \bigsqcup \boxed{C}_\ell^I$ will join

⁵ Note the non- \perp underlying effect; well-typed expressions do not have degenerate effects.

the underlying effect of all continuations invoked by the body. T-PROMPT also performs some checking of result types (which all hold trivially since all types involved are `unit`), and prophecy validity checks that yield constraints we can solve to derive a closed-form type rule for the loop.

Completing a typing derivation with final underlying effect $Q_\ell = Q_e \sqcup \bigsqcup [C_{\ell}^r]_\ell^I$ is possible given the solutions to the effect-related constraints imposed by `validEffects`: $\perp \sqsubseteq Q_p$, and $(Q_e \triangleright [C_p]_\ell) \cup ((Q_e \triangleright Q_p) \triangleright \{\text{replace } \ell : I \rightsquigarrow \text{unit}\}) \sqsubseteq C_p$. These could be read off a hypothetical derivation (see appendices in the technical report [32]) yielding the derived rule

$$\frac{\Gamma \vdash e : \tau \mid (\emptyset, \emptyset, Q_e) \quad \perp \sqsubseteq Q_p \quad (Q_e \triangleright [C_p]_\ell) \cup ((Q_e \triangleright Q_p) \triangleright \{\text{replace } \ell : I \rightsquigarrow \text{unit}\}) \sqsubseteq C_p}{\Gamma \vdash [\text{loop } e] : \text{unit} \mid (\emptyset, \emptyset, Q_e \sqcup \bigsqcup [C_{\ell}^r]_\ell^I)}$$

However, this rule is more complex than we would like for a simple infinite loop (note we have not expanded $C = (Q_e \triangleright [C_p]_\ell) \cup ((Q_e \triangleright Q_p) \triangleright \{\text{replace } \ell : I \rightsquigarrow \text{unit}\})$), and also exposes details of the continuation-aware effects – which is undesirable if the goal is to derive closed rules for using the loop by itself, without developer access to full continuations, and there is an additional requirement that the prophecy used in the derivation is non-trivial (from T-PROMPT). These constraints can be satisfied by $Q_p = Q_e^*$ (thus not \perp , ensuring a non-trivial prophecy), with $C_p = \{\text{replace } \ell : Q_e^* \rightsquigarrow \text{unit}\}$ (so $[C_p]_\ell = C_p$). The choice for C_p ensures that any “unrolling” of the loop to include any number of Q_e prefixes (as generated by the left operand of the union in the last constraint) is in fact less than the replacement effect ($Q_e \triangleright Q_e^* \sqsubseteq Q_e^*$). This then implies that $Q_e \sqcup \bigsqcup [C_{\ell}^r]_\ell^I \sqsubseteq Q_e \sqcup (Q_e^*) \sqsubseteq Q_e^*$, by properties of Gordon’s iteration operator [30, 31] (Section 2). Assuming $cc \notin \Gamma$ (or hygienic macros) and applying subsumption, this leads us to the pleasingly simple derived rule:

$$\text{D-INFLLOOP} \frac{\Gamma \vdash e : \tau \mid (\emptyset, \emptyset, Q_e)}{\Gamma \vdash [\text{loop } e] : \text{unit} \mid (\emptyset, \emptyset, Q_e^*)}$$

7.2 Other Derived Rules

We have similarly macro-expressed traditional while loops, exceptions (`try-catch` and `throw`), and generators [14] (a form of coroutine now common in C#, Python, and JavaScript), and derived rules for each. Figure 6 gives derived rules for these constructs under a variety of conditions, each derived following a similar process to D-INFLLOOP.

Loops do not *require* the use of control operators to express, of course. However, they are an important consistency check. Notably, D-WHILE recovers, via the approach above, exactly the rule for while loops that was hand-crafted in prior work [25, 26] and only recently given general treatment [30, 31]. D-ABORTINGWHILE is a limited generalization of D-WHILE, for the case where the condition or body by use `abort` (e.g., throw exceptions). The underlying effect – for when the loop completes normally – is as in D-WHILE. The control effect reflects the various times an abort may be thrown, based on the assumptions about aborts in c and e : during the initial execution of the condition, the initial execution of the body, or by *subsequent* executions of the condition or body. The derived rules for `try-catch` and `throw` reflect their simple expression in terms of prompts and `abort`.

The generator rule is more complex, requiring a bit of careful thought about the well-formedness condition `GenProphs`, but still following the general approach taken for D-LOOP. Recall that generators are a construct for producing some sequence of values asynchronously: a generator is an object that may be queried repeatedly, and each query either produces a new value or indicates there are no more values. This is similar to an iterator, except generators are written in direct style. A language or library typically exposes a `yield` construct to

23:24 Lifting Sequential Effects to Control Operators

$$\begin{array}{c}
\text{D-WHILE} \\
\frac{\Gamma \vdash e : (\emptyset, \emptyset, Q_e) \quad \Gamma \vdash c : (\emptyset, \emptyset, Q_c)}{\Gamma \vdash \llbracket \text{while } c \ e \rrbracket : \text{unit} \mid (\emptyset, \emptyset, Q_c \triangleright (Q_e \triangleright Q_c)^*)} \\
\text{D-FULLINFLLOOP} \\
\frac{\Gamma \vdash e : \tau_e \mid \chi_e}{\Gamma \vdash \llbracket \text{loop } e \rrbracket : \text{unit} \mid \chi_e^*} \\
\text{D-ABORTINGWHILE} \\
\frac{l \notin C_c \quad l \notin C_e \quad \text{AbortsOnly}(C_c \cup C_e) \quad \Gamma \vdash c : (\emptyset, C_c, Q_c) \quad \Gamma \vdash e : (\emptyset, C_e, Q_e)}{\Gamma \vdash \llbracket \text{while}_\ell c \ e \rrbracket : \text{unit} \mid (\emptyset, \bigcup \left\{ \begin{array}{l} C_c \cup (Q_c \triangleright Q_e \triangleright C_c) \cup (Q_c \triangleright (Q_e \triangleright Q_c)^* \triangleright C_e), \\ (Q_c \triangleright (Q_e \triangleright Q_c)^* \triangleright Q_e \triangleright C_c) \end{array} \right\}, Q_c \triangleright (Q_e \triangleright Q_c)^*)} \\
\text{D-TRYCATCH} \\
\frac{\Gamma \vdash e : \tau \mid (\emptyset, \{\text{abort } \ell_C \ Q \rightsquigarrow C\}, Q_e) \quad \Gamma \vdash h : C \xrightarrow{(\emptyset, \emptyset, Q_h)} \tau \mid (\emptyset, \emptyset, I)}{\Gamma \vdash \llbracket \text{try } e \ \text{catch } C \Rightarrow h \rrbracket : \tau \mid (\emptyset, \emptyset, Q_e \sqcup (Q \triangleright Q_h))} \quad \text{D-THROW} \\
\frac{\Gamma \vdash e : C \mid (\emptyset, \emptyset, Q_e)}{\Gamma \vdash \llbracket \text{throw}_C \ e \rrbracket : \tau \mid (\emptyset, \{\text{abort } \ell_C \ Q_e \rightsquigarrow C\}, I)} \\
\text{GenProphs}(P, \ell, E) = \frac{\forall \ell', P', C', Q', P'', C'', Q'', \tau. \quad \text{prophecy } \ell' (P', C', Q') \rightsquigarrow \tau \ \text{obs } (P'', C'', Q'') \in P \Rightarrow \ell' = \ell \wedge \tau = \text{bool} \wedge \text{GenProphs}(P', \ell, E) \wedge C' \sqsubseteq \{\text{abort } \ell \ E^* \rightsquigarrow (\text{option } \tau)\} \wedge Q' \sqsubseteq E^* \wedge \lceil \overline{P'} \rceil \sqsubseteq \lceil \overline{P''} \rceil \wedge C'' \sqsubseteq C' \wedge Q'' \sqsubseteq Q'}{\Gamma \vdash f : \left(\tau \xrightarrow{(\{\text{prophecy } \text{gen } (P_p, C_p, E^*) \rightsquigarrow (\text{option } \tau) \ \text{obs } (\emptyset, \emptyset, I)\}, \{\text{abort } \text{gen } I \rightsquigarrow (\text{option } \tau)\}, \perp)} \text{unit}) \xrightarrow{I} \right) \mid (\emptyset, \emptyset, I)} \\
\frac{C \sqsubseteq \{\text{abort } \text{gen } (E^*) \rightsquigarrow (\text{option } \tau)\} \quad \text{GenProphs}(P, \text{gen}, E)}{\Gamma \vdash \llbracket \text{iterate } \text{init } \text{gen } f \rrbracket : \text{unit} \xrightarrow{(\emptyset, \emptyset, E^*)} \text{option } \tau \mid (\emptyset, \emptyset, I)}
\end{array}$$

■ **Figure 6** A collection of derived rules.

produce a value for the client. (This now describes Python, C#, F#, and JavaScript, among other languages.) After the client consumes each value and queries the generator again, control resumes immediately after the last-executed `yield`, continuing until another value is `yielded` or generation is complete. Consider a simple client of Coyle and Crogono’s implementation using Scheme macros and `call/cc` [14] (see our technical report for Racket code [32]):

```
(define next (iterate (lambda (yield done) (yield "a") (yield "b") (done))))
```

the iterator body takes two arguments, `yield` and `done` for yielding a value, and indicating generation is complete respectively. `next` is the result of building a generator from this function; `iterate` (the generator implementation) supplies functions for `yield` and `done` (described below).

Our macro `iterate init gen f` (`init` and `gen` are tags) expands to a heap-storage-based generator encoding similar to Coyle and Crogono’s [14] (and also given explicitly with type annotations in our technical report [32]). A prompt is used to delimit the scope of the generator body. Yielding a value captures the current continuation up to that prompt, stores it in a heap cell, then uses `abort` to throw the yielded value. The rule D-ITERATE assumes `f` is a function of two arguments as above: the first is a function playing the role of `yield`, the second a function that marks the iterator as complete (`done`). The macro returns a function which, when invoked, returns an option containing either the next element produced by the generator, or a failure. When invoked, this function introduces a new prompt for tag `gen`, and invokes the stored continuation to produce the next element (via the `yield` parameter) or a completion. The rule assumes the activity “between” successive yields can be over-approximated by an underlying effect `E`, and stores the continuations with underlying effect `E*`. Because uses of the first parameter, `yield`, capture continuations, a prophecy choice must be made (P_p) for that particular generator, for the yield to predict

the appropriate remainder of the generator body. D-ITERATE is specialized here to the assumption that all prophecies and aborts in the generator body are related to the tag *gen* only, which allows us to lift the requirements of prophecy validation into the `GenProphs` predicate. A complete rule permitting at least exceptions to escape a generator is possible, but would be very complex (so much so that C# strongly restricts their interactions [53]).

8 Related Work

Recent years have seen great progress on semantic models for sequential effect systems [70, 42, 55], centering on what are now known as *graded monads*: monads indexed by some kind of monoid (to model sequential composition), commonly a partially-ordered monoid following Katsumata [42]. Gordon [30] focused on capturing common structure for prior concrete effect systems, leading to the first abstract characterization of sequential effect systems with singleton effects, effect polymorphism, and iteration of sequential effects.

To the best of our knowledge we are the first to use the term “accumulator” as we do to identify this as a reusable technique. However accumulators have appeared before. Koskinen and Terauchi’s effect system [45] uses left-accumulators for safety and liveness properties (requiring an oracle for liveness). Effects in their system are a pair of sets, one a set of finite traces (for terminating executions) and the other a set of infinite traces (for non-terminating executions). The infinite traces left-accumulate: code that comes after a non-terminating expression in program-order never runs. On the other hand, *finite* executions from code *before* an infinite execution extend the prefix of the infinite executions. Earlier, Neamtiu et al. [56] defined *contextual* effects to track what (otherwise order-unaware) effects occurred before or after an expression, to ensure key correctness properties for code using dynamic software updates.

Effect systems treating continuations are nearly as old as effect systems themselves [41]. To the best of our knowledge, we are the first to integrate *sequential* effects with exceptions, generators, or general delimited continuations – or any control flow construct beyond while loops, including any form of continuation, tagged or otherwise. As mentioned in Section 2, the original motivation for tags was to prevent encodings of separate control operators from interfering with each other [64], which is critical for our goals, strictly more expressive than untagged continuations, and motivates important elements of the theory (blocking). The only other work we know of focusing on effects with tagged delimited continuations is Pretnar and Bauer’s variant [62] of algebraic effects and handlers [62] where operations may be handled by outer `handle` constructs (not just the closest construct as in other algebraic effects work). Their commutative effects ensures all algebraic operations are handled by some enclosing handler.

Tov and Pucella [73] examined the interaction of *untagged* delimited continuations with substructural types (a coeffect [58]). Delbianco and Nanevski adapted Hoare Type Theory for untagged *algebraic* continuations [17], which include prompts and handlers, but place handlers at the site of an abort rather than at the prompt in order to satisfy some useful computational equalities (see below). As a consequence, encoding non-trivial control flow constructs in their system becomes significantly more complex; for example, simulating the standard semantics of throwing exceptions to the nearest enclosing catch block for the exception type would require catching, dispatching, and re-throwing at every prompt. This and lack of tagging would make compositional study of multiple control flow constructs / control operators difficult, and as our work shows the treatment of multiple tags is not a trivial extension of untagged semantics. Atkey [6] considered denotational semantics for (untagged)

composable continuations in his parameterized monad framework for (denotational) sequential effect systems, essentially giving a denotation of a type-and-effect system for answer type modification [5, 16] – a kind of sequential effect which can be used to allow continuations to temporarily change the result type of a continuation, as long as it is known (via the effects) that it will be changed back. Thus Atkey considered answer type modification effects as an instance of a sequential effect specific to using control operators, rather than having an application-domain-focused effect (like exceptions or traces) work with continuations or giving an account of general sequential effects for control operators. Readers familiar with answer type modification may wonder about directly supporting it in our generic core language. We have not yet considered this deeply, but note that (i) directly ascribing answer type modifications to control operations would require assigning *specific* answer type modification effects to the control operations, not just effects derived from primitives, but (ii) Kobori et al. [44] showed that tagged shift/reset can express answer type modification in a type system that does not track answer types explicitly, so such an extension may provide no additional power (treating convenience as another matter).

Algebraic effects with handlers [60] are a means to describe the *semantics* of effects in terms of a set of operations (the effectful operations) along with handlers that interpret those operations as actions on some resource. The combination yields an algebra characterizing equality of different effectful program expressions, hence the term “algebraic”. Languages with algebraic effects include an effect system to reason about which effects a computation uses, to ensure they are handled. Some implementations even use Lindley and Cheney’s effect adaptation [49] of row polymorphism [75] to support effect inference [47]. Handlers for algebraic effects receive both the action to interpret and the continuation of the program following the effectful action. Thus they can implement many control operators, including generators and cooperative multithreading [48], as with the delimited continuations we study. In an untyped setting without tagging, algebraic handlers can simulate (via macro translation) `shift0/reset0` [28], which can simulate prompts and handlers [63] (with correct space complexity, not only extensionally-correct behavior); with those limitations, handlers are as powerful as the constructs we study. For the common commutative effect system for handlers that ensures all operations are handled, Forster et al. [28] prove that the translation from handlers to prompts (`shift0`) is not type-and-effect preserving, and conjecture the reverse translation also fails to preserve types. They conjectured that adding polymorphism to each system would enable a type-and-effect preserving translation (again, without tagging, for a commutative effect system), which was recently confirmed by Piróg et al. [59] for a class of commutative effect systems.

The effect systems considered for algebraic effects thus far have only limited support for reasoning about sequential effects. The types given for individual algebraic effects do support reasoning about the existence of a certain type of resource before and after the computation [12, 7]. However, the way this is done corresponds to a parameterized monad [6], which Tate showed [70] crisply do not include all meaningful sequential effect systems. His examples that cannot be modeled as parameterized monads include examples that *can* be modeled as effect quantales [31], such as the non-reentrant locking effect system Tate uses to motivate aspects of his approach.

General considerations of sequential effect systems have not yet been explored for algebraic effects. When it is considered, it seems likely ideas from our development (particularly prophecies) will be useful. For example, Dolan et al. [18] offer two reasons for dynamically enforcing linearity of continuations in their handlers: performance, but also avoiding the sorts of errors prevented by sequential effect systems, such as closing a file twice by reusing a continuation.

It also seems plausible that our approach could be adapted to algebraic effects and handlers. With an effectively-tagged version of handlers [62], a similar macro-expression of control flow constructs and control operators is likely feasible, in particular adapting our notion of prophecy and observation to handlers: in this case, the continuations themselves are seen by handlers that are direct subexpressions of the handling construct itself, so prophecies might observe “outside-in” rather than our system’s “inside-out” accumulation.

The approach we take to deriving type rules for control flow constructs and control operators is reminiscent of work done in parallel with ours by Pombrio and Krishnamurthi [61]. They address the problem of producing useful type rules when a language semantics and type rules are defined directly for a simpler core language, and a full source language is defined using syntactic sugar (i.e., macros) that expand into core language expressions with the intended semantics, such as the approach taken by λ_{JS} [37]. There the issue is that type errors given in terms of the elaborated core terms are difficult to understand for developers writing in the unelaborated source language. Pombrio and Krishnamurthi offer an approach to automatically lift core language type rules through the desugaring process to the source language, providing sensible source-level type errors. Their work focuses on type systems without effects, but including such notions as subtyping and existential types. They do not consider control operators (delimited continuations) or effects (neither commutative nor sequential). Extending their approach to support the language features and types (effects) we consider would make our approach more useful to effect system designers, though this is non-trivial due to the many ways to combine sequential effects.

9 Conclusions

We have given the first general approach to integrating arbitrary sequential effect systems with tagged delimited control operators, which allows lifting existing sequential effect systems without knowledge of control operators to automatically support tagged delimited control. We have used this characterization to derive sequential effect system rules for standard control flow structures macro-expressed via continuations, including deriving known forms (loops) and giving the first characterization of exceptions and generators in sequential effect systems.

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, //month = mar), 2006.
- 2 Martín Abadi and Leslie Lamport. The existence of refinement mappings. In *LICS*, 1988.
- 3 Martín Abadi and Leslie Lamport. The existence of refinement mappings. *Theoretical Computer Science*, 82(2):253–284, 1991.
- 4 Torben Amtoft, Flemming Nielson, and Hanne Riis Nielson. *Type and Effect Systems: Behaviours for Concurrency*. Imperial College Press, London, UK, 1999.
- 5 Kenichi Asai and Yukiyooshi Kameyama. Polymorphic delimited continuations. In *APLAS*, pages 239–254, 2007.
- 6 Robert Atkey. Parameterised Notions of Computation. *Journal of Functional Programming*, 19:335–376, July 2009.
- 7 Andrej Bauer and Matija Pretnar. An effect system for algebraic effects and handlers. In *International Conference on Algebra and Coalgebra in Computer Science*, pages 1–16. Springer, 2013.
- 8 Garrett Birkhoff. *Lattice theory*, volume 25 of *Colloquium Publications*. American Mathematical Soc., 1940. Third edition, eighth printing with corrections, 1995.

- 9 Thomas Scott Blyth. *Lattices and ordered algebraic structures*. Springer Science & Business Media, 2006.
- 10 Chandrasekhar Boyapati, Robert Lee, and Martin Rinard. Ownership Types for Safe Programming: Preventing Data Races and Deadlocks. In *OOPSLA*, 2002.
- 11 Chandrasekhar Boyapati and Martin Rinard. A Parameterized Type System for Race-Free Java Programs. In *OOPSLA*, 2001.
- 12 Edwin Brady. Programming and reasoning with algebraic effects and dependent types. In *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming, ICFP '13*, pages 133–144. ACM, 2013. doi:10.1145/2500365.2500581.
- 13 John Clements, Matthew Flatt, and Matthias Felleisen. Modeling an algebraic stepper. In *European symposium on programming*, pages 320–334. Springer, 2001.
- 14 Christopher Coyle and Peter Crogono. Building abstract iterators using continuations. *SIGPLAN Not.*, 26(2):17–24, January 1991. doi:10.1145/122179.122181.
- 15 Olivier Danvy. An analytical approach to program as data objects, 2006. DSc thesis, Department of Computer Science, Aarhus University.
- 16 Olivier Danvy and Andrzej Filinski. A functional abstraction of typed contexts. Technical report, DIKU — Computer Science Department, University of Copenhagen, July 1989.
- 17 Germán Andrés Delbianco and Aleksandar Nanevski. Hoare-style reasoning with (algebraic) continuations. In *ICFP*, 2013.
- 18 Stephen Dolan, Spiros Eliopoulos, Daniel Hillerström, Anil Madhavapeddy, K. C. Sivaramakrishnan, and Leo White. Concurrent system programming with effect handlers. In Meng Wang and Scott Owens, editors, *Trends in Functional Programming*, pages 98–117, Cham, 2018. Springer International Publishing.
- 19 Matthias Felleisen. The theory and practice of first-class prompts. In *Conference Record of the Fifteenth Annual ACM Symposium on Principles of Programming Languages, San Diego, California, USA, January 10-13, 1988*, pages 180–190, 1988. doi:10.1145/73560.73576.
- 20 Matthias Felleisen. On the expressive power of programming languages. *Science of computer programming*, 17(1-3):35–75, 1991.
- 21 Matthias Felleisen, Robert Bruce Findler, and Matthew Flatt. *Semantics engineering with PLT Redex*. Mit Press, 2009.
- 22 Matthias Felleisen and Daniel P. Friedman. A reduction semantics for imperative higher-order languages. In *PARLE, Parallel Architectures and Languages Europe, Volume II: Parallel Languages, Eindhoven, The Netherlands, June 15-19, 1987, Proceedings*, pages 206–223, 1987. doi:10.1007/3-540-17945-3_12.
- 23 Cormac Flanagan and Martín Abadi. Object Types against Races. In *CONCUR*, 1999.
- 24 Cormac Flanagan and Martín Abadi. Types for Safe Locking. In *ESOP*, 1999.
- 25 Cormac Flanagan and Shaz Qadeer. A type and effect system for atomicity. In *PLDI*, 2003.
- 26 Cormac Flanagan and Shaz Qadeer. Types for atomicity. In *TLDI*, 2003.
- 27 Matthew Flatt, Gang Yu, Robert Bruce Findler, and Matthias Felleisen. Adding delimited and composable control to a production programming environment. In *ICFP*, 2007.
- 28 Yannick Forster, Ohad Kammar, Sam Lindley, and Matija Pretnar. On the expressive power of user-defined effects: Effect handlers, monadic reflection, delimited control. *Proc. ACM Program. Lang.*, 1(ICFP):13:1–13:29, August 2017. doi:10.1145/3110257.
- 29 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.
- 30 Colin S. Gordon. A Generic Approach to Flow-Sensitive Polymorphic Effects. In *ECOOP*, 2017.
- 31 Colin S. Gordon. Polymorphic Iterable Sequential Effect Systems. Technical Report arXiv cs.PL/cs.LO 1808.02010, Computing Research Repository (Corr), August 2018. In Submission.. arXiv:1808.02010.

- 32 Colin S. Gordon. Sequential Effect Systems with Control Operators. Technical Report arXiv cs.PL 1811.12285, Computing Research Repository (CoRR), December 2018. arXiv: 1811.12285.
- 33 Colin S. Gordon, Werner Dietl, Michael D. Ernst, and Dan Grossman. JavaUI: Effects for Controlling UI Object Access. In *ECOOP*, 2013.
- 34 Colin S. Gordon, Michael D. Ernst, and Dan Grossman. Static Lock Capabilities for Deadlock Freedom. In *TLDI*, 2012.
- 35 James Gosling, Bill Joy, Guy L Steele, Gilad Bracha, and Alex Buckley. *The Java Language Specification: Java SE 8 Edition*. Pearson Education, 2014.
- 36 OpenJDK HotSpot Group. OpenJDK Project Loom: Fibers and Continuations, 2019. URL: <https://wiki.openjdk.java.net/display/loom/Main>.
- 37 Arjun Guha, Claudiu Saftoiu, and Shriram Krishnamurthi. The Essence of JavaScript. In *ECOOP*, 2010.
- 38 Christopher T. Haynes and Daniel P. Friedman. Embedding continuations in procedural objects. *ACM Trans. Program. Lang. Syst.*, 9(4):582–598, October 1987. doi:10.1145/29873.30392.
- 39 Christopher T. Haynes, Daniel P. Friedman, and Mitchell Wand. Continuations and coroutines. In *LISP and Functional Programming*, pages 293–298, 1984.
- 40 Christopher T. Haynes, Daniel P. Friedman, and Mitchell Wand. Obtaining coroutines with continuations. *Comput. Lang.*, 11(3/4):143–153, 1986. doi:10.1016/0096-0551(86)90007-X.
- 41 P. Jouvelot and D. K. Gifford. Reasoning about continuations with control effects. In *PLDI*, 1989.
- 42 Shin-ya Katsumata. Parametric effect monads and semantics of effect systems. In *POPL*, 2014.
- 43 Casey Klein, John Clements, Christos Dimoulas, Carl Eastlund, Matthias Felleisen, Matthew Flatt, Jay A. McCarthy, Jon Raskind, Sam Tobin-Hochstadt, and Robert Bruce Findler. Run your research: On the effectiveness of lightweight mechanization. In *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '12, pages 285–296, New York, NY, USA, 2012. ACM. doi:10.1145/2103656.2103691.
- 44 Ikuo Kobori, Yukiyoshi Kameyama, and Oleg Kiselyov. Answer-type modification without tears: Prompt-passing style translation for typed delimited-control operators. In Olivier Danvy and Ugo de'Liguoro, editors, *Proceedings of the Workshop on Continuations, WoC 2016, London, UK, April 12th 2015*, volume 212 of *EPTCS*, pages 36–52, 2015. doi:10.4204/EPTCS.212.3.
- 45 Eric Koskinen and Tachio Terauchi. Local temporal reasoning. In *CSL/LICS*, 2014.
- 46 Shriram Krishnamurthi, Peter Walton Hopkins, Jay A. McCarthy, Paul T. Graunke, Greg Pettyjohn, and Matthias Felleisen. Implementation and use of the PLT scheme web server. *Higher-Order and Symbolic Computation*, 20(4):431–460, 2007. doi:10.1007/s10990-007-9008-y.
- 47 Daan Leijen. Koka: Programming with Row Polymorphic Effect Types. In *Proceedings 5th Workshop on Mathematically Structured Functional Programming (MSFP)*, 2014.
- 48 Daan Leijen. Structured asynchrony with algebraic effects. In *Proceedings of the 2nd ACM SIGPLAN International Workshop on Type-Driven Development*, TyDe 2017, pages 16–29, New York, NY, USA, 2017. ACM. doi:10.1145/3122975.3122977.
- 49 Sam Lindley and James Cheney. Row-based effect types for database integration. In *Proceedings of the 8th ACM SIGPLAN workshop on Types in language design and implementation*, pages 91–102. ACM, 2012.
- 50 J. M. Lucassen and D. K. Gifford. Polymorphic Effect Systems. In *POPL*, 1988.
- 51 Daniel Marino and Todd Millstein. A Generic Type-and-Effect System. In *TLDI*, 2009. doi:10.1145/1481861.1481868.
- 52 Microsoft. C# Language Specification: Enumerable Objects, 2018. URL: <https://github.com/dotnet/csharp-lang/blob/master/spec/classes.md#enumerable-objects>.
- 53 Microsoft. C# Reference: yield Exception Handling, 2018. URL: <https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/yield#exception-handling>.

- 54 Mozilla. Mozilla Developer Network Documentation: `function*`, 2018. URL: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/function*.
- 55 Alan Mycroft, Dominic Orchard, and Tomas Petricek. Effect systems revisited — control-flow algebra and semantics. In *Semantics, Logics, and Calculi*. Springer, 2016.
- 56 Iulian Neamtiu, Michael Hicks, Jeffrey S Foster, and Polyvios Pratikakis. Contextual effects for version-consistent dynamic software updating and safe concurrent programming. In *POPL*, pages 37–49, 2008.
- 57 Flemming Nielson and Hanne Riis Nielson. From cml to process algebras. In *CONCUR*, 1993.
- 58 Tomas Petricek, Dominic Orchard, and Alan Mycroft. Coeffects: A calculus of context-dependent computation. In *ICFP*, 2014.
- 59 Maciej Piróg, Piotr Polesiuk, and Filip Sieczkowski. Typed equivalence of effect handlers and delimited control. In *4th International Conference on Formal Structures for Computation and Deduction (FSCD 2019)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2019.
- 60 Gordon Plotkin and Matija Pretnar. Handlers of algebraic effects. In *European Symposium on Programming*, pages 80–94. Springer, 2009.
- 61 Justin Pombrio and Shriram Krishnamurthi. Inferring type rules for syntactic sugar. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2018, pages 812–825, New York, NY, USA, 2018. ACM. doi:10.1145/3192366.3192398.
- 62 Matija Pretnar and Andrej Bauer. An effect system for algebraic effects and handlers. *Logical Methods in Computer Science*, 10, 2014.
- 63 Chung-chieh Shan. A static simulation of dynamic delimited control. *Higher-Order and Symbolic Computation*, 20(4):371–401, 2007.
- 64 Dorai Sitaram. Handling control. In *PLDI*, 1993.
- 65 Dorai Sitaram and Matthias Felleisen. Control delimiters and their hierarchies. *Lisp and Symbolic Computation*, 3(1):67–99, 1990.
- 66 Dorai Sitaram and Matthias Felleisen. Reasoning with continuations II: full abstraction for models of control. In *LISP and Functional Programming*, pages 161–175, 1990. doi:10.1145/91556.91626.
- 67 Christian Skalka. Types and trace effects for object orientation. *Higher-Order and Symbolic Computation*, 21(3):239–282, 2008.
- 68 Christian Skalka, Scott Smith, and David Van Horn. Types and trace effects of higher order programs. *Journal of Functional Programming*, 18(2), 2008.
- 69 Kohei Suenaga. Type-based deadlock-freedom verification for non-block-structured lock primitives and mutable references. In *APLAS*, 2008.
- 70 Ross Tate. The sequential semantics of producer effect systems. In *POPL*, 2013.
- 71 Python Development Team. Python Enhancement Proposal 255: Simple Generators, 2001. URL: <https://www.python.org/dev/peps/pep-0255/>.
- 72 Mads Tofte and Jean-Pierre Talpin. Region-based memory management. *Information and computation*, 132(2):109–176, 1997.
- 73 Jesse A. Tov and Riccardo Pucella. A theory of substructural types and control. In *OOPSLA*, 2011.
- 74 Philip Wadler and Peter Thiemann. The Marriage of Effects and Monads. *Transactions on Computational Logic (TOCL)*, 4:1–32, 2003.
- 75 Mitchell Wand. Type inference for record concatenation and multiple inheritance. In *Logic in Computer Science, 1989. LICS'89, Proceedings., Fourth Annual Symposium on*, pages 92–97. IEEE, 1989.

Flow-Sensitive Type-Based Heap Cloning

Mohamad Barbar

University of Technology Sydney, Australia
CSIRO's Data61, Sydney, Australia

Yulei Sui¹

University of Technology Sydney, Australia

Shiping Chen

CSIRO's Data61, Sydney, Australia

Abstract

By respecting program control-flow, flow-sensitive pointer analysis promises more precise results than its flow-insensitive counterpart. However, existing heap abstractions for C and C++ flow-sensitive pointer analyses model the heap by creating a single abstract heap object for each memory allocation. Two runtime heap objects which originate from the same allocation site are imprecisely modelled using one abstract object, which makes them share the same imprecise points-to sets and thus reduces the benefit of analysing heap objects flow-sensitively. On the other hand, equipping flow-sensitive analysis with context-sensitivity, whereby an abstract heap object would be created (cloned) per calling context, can yield a more precise heap model, but at the cost of uncontrollable analysis overhead when analysing larger programs.

This paper presents TYPECLONE, a new type-based heap model for flow-sensitive analysis. Our key insight is to differentiate concrete heap objects lazily using type information at use sites within the program control-flow (e.g., when accessed via pointer dereferencing) for programs which conform to the strict aliasing rules set out by the C and C++ standards. The novelty of TYPECLONE lies in its lazy heap cloning: an untyped abstract heap object created at an allocation site is killed and replaced with a new object (i.e. a clone), uniquely identified by the type information at its use site, for flow-sensitive points-to propagation. Thus, heap cloning can be performed within a flow-sensitive analysis without the need for context-sensitivity. Moreover, TYPECLONE supports new kinds of strong updates for flow-sensitive analysis where heap objects are filtered out from imprecise points-to relations at object use sites according to the strict aliasing rules. Our method is neither strictly superior nor inferior to context-sensitive heap cloning, but rather, represents a new dimension that achieves a sweet spot between precision and efficiency. We evaluate our analysis by comparing TYPECLONE with state-of-the-art sparse flow-sensitive points-to analysis using the 12 largest programs in GNU Coreutils. Our experimental results also confirm that TYPECLONE is more precise than flow-sensitive pointer analysis and is able to, on average, answer over 15% more alias queries with a no-alias result.

2012 ACM Subject Classification Software and its engineering → Automated static analysis

Keywords and phrases Heap cloning, type-based analysis, flow-sensitivity

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2020.24

Supplementary Material ECOOP 2020 Artifact Evaluation approved artifact available at
<https://doi.org/10.4230/DARTS.6.2.1>.
<https://github.com/SVF-tools/SVF/wiki/TypeClone>

Funding This research is supported by Australian Research Grant DP200101328.
Mohamad Barbar: supported by a PhD scholarship funded by CSIRO's Data61.

Acknowledgements We would like to thank the anonymous reviewers for their helpful comments.

¹ Corresponding author



© Mohamad Barbar, Yulei Sui, and Shiping Chen;
licensed under Creative Commons License CC-BY

34th European Conference on Object-Oriented Programming (ECOOP 2020).

Editors: Robert Hirschfeld and Tobias Pape; Article No. 24; pp. 24:1–24:26

Leibniz International Proceedings in Informatics



LIPICs Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany



1 Introduction

Pointer analysis aims to determine the objects which a pointer may point to at runtime. It is an enabling technology which forms a basis for other program analysis tasks such as compiler optimisation [28, 13], program slicing [42, 43], enforcing control-flow integrity [23, 16], and software security analysis [38, 32].

1.1 Background

Flow-sensitivity is an essential precision dimension of pointer analysis. Unlike flow-insensitive analysis which considers instructions to be unordered, flow-sensitive analysis accounts for program execution order to better approximate runtime behaviour and achieve a more precise result. Traditional flow-sensitive points-to analysis computes and maintains points-to relations at each program point. These points-to relations are propagated along the program's control-flow graph (CFG) by solving an iterative data-flow problem until a fixed point is reached.

In recent years, there have been significant advances in making flow-sensitive analysis for C and C++ programs more efficient. Rather than propagating all relations to/from each program point on the CFG, SPARSE analysis [19, 20], a flow-sensitive and context-insensitive analysis, pre-computes over-approximated value-flows (def-use chains) to produce a value-flow graph upon which the main phase analysis can propagate points-to relations sparsely. This reduces both time and space overhead while maintaining precision. Selective flow-sensitive analysis [30] performs strong updates for stack and global variables at stores where flow-sensitive singleton points-to sets are available, but falls back to flow-insensitive results otherwise, making a trade between efficiency and precision. SPARSE analysis incorporates these strong updates. As an alternative to whole-program analysis, demand-driven flow-sensitive analysis [39] aims to answer points-to queries flow-sensitively by only analysing specific parts of a program with CFL-reachability on the pre-computed value-flow graph.

1.2 Motivation and insights

Most recent advances in C/C++ flow-sensitive pointer analysis focus on improving efficiency. Apart from those which employ expensive context-sensitivity, existing solutions exclusively use an *allocation-site-based* heap abstraction where an abstract object is created per memory allocation site to represent the set of concrete objects created at that allocation site during runtime. This is especially coarse for heap-intensive programs when allocation sites are contained within allocation wrapper functions [7] since any pointers pointing to objects originating from such wrapper functions will be regarded as having a may-alias relation despite originating in different contexts at runtime. Thus the heap abstraction significantly reduces the benefit of flow-sensitivity by restricting most of the precision improvement to stack and global variables only. For example, given two concrete heap objects o_1 and o_2 which originate from the same allocation wrapper and are accessed along two different control-flow branches, the flow-sensitive points-to results of the heap objects, and of any pointers which point to them, would be as (im)precise as flow-insensitive results since the heap abstraction would treat both concrete objects as a single abstract object. Whenever a single allocation wrapper is used exclusively, the heap abstraction is as precise as having one abstract object represent all concrete heap objects allocated by the program.

Developing a precise and efficient flow-sensitive heap abstraction is challenging since the infinite-sized heap needs to be partitioned into a finite a number of abstract objects. A straightforward solution is to equip flow-sensitive analysis with context-sensitivity. A

context-sensitive analysis performs heap cloning based on calling contexts: a new heap object (a clone) is created for each calling context reaching the object's allocation site. However, given the billions of calling contexts in large programs [44], adding context-sensitivity to flow-sensitive analysis incurs uncontrollable overhead even with *k-limiting* enabled [3].

Rather than heap cloning according to calling context, we would like to investigate the use of type information and the initialisation of an object's type (e.g. through dereferencing a pointer to that object) to perform heap cloning. Our key insight is based on the strict aliasing rules laid out by the C [2, §6.5 ¶7] and C++ [1, §6.10 ¶8] standards whereby a pointer may only access an object with a compatible type (or else, the program exhibits undefined behaviour). For example, reading an object of type `float` through a pointer of type `int *` is undefined behaviour. Thus, a pointer of type `int *` and a pointer of type `float *`, for example, cannot be referring to the same object when being dereferenced. We can then use such type information to “separate” concrete objects from within an abstract object (of the allocation-site-based abstraction) into multiple abstract objects, each used for accesses of a specific type. More concretely, an untyped heap object *o* can be cloned into multiple typed objects based upon the accesses (or dereferences) of *o*. Therefore, type-based heap cloning could more precisely distinguish heap objects while avoiding expensive context-sensitive heap modeling.

1.3 Existing efforts and limitations

Type-based heap modelling [24, 26, 37] has been used in strongly typed languages, such as Java, to produce a cost-effective heap abstraction for context-sensitive but flow-insensitive pointer analysis. However, there are very few attempts at type-based pointer analysis for C and C++ [5, 7, 13]. C and C++ introduce new challenges:

- C and C++ are weakly typed so reasoning about the types of objects, especially heap objects, is difficult.
- High-level C/C++ type information is not preserved in the intermediate representation (IR) of modern compilers, like Clang/LLVM [14], upon which static analyses typically operate on.
- C and C++ allow for the address of fields to be taken and for pointer arithmetic within an object, which means fields' types must too be resolved during the analysis.

It is hard to design a fully sound pointer analysis, incorporating types, for non-conforming programs which violate the strict aliasing rules. Generally, a fully sound analysis would be unscalable or imprecise almost to the point of uselessness on its own [31]. Type-based alias analysis (TBAA) [13] made the initial attempt at exploiting the strict aliasing assumption (or its equivalent in Modula-3) to produce a fast alias analysis. The almost stateless nature of TBAA makes it especially useful in resolving alias queries that would otherwise require inter-procedural analysis to non-trivially answer. Modern compilers like LLVM and GCC implement TBAA behind the `-fstrict-aliasing` flag (enabled by default). In reality, programs which wish to safely benefit from optimisations enabled by TBAA must conform to the strict aliasing rules. Programs which fail to do so either risk miscompilation (since the program exhibits undefined behaviour the compiler assumes is not present) or must notify the compiler that the program violates the strict aliasing rules and do without those optimisations. Violating these rules (or invoking any other undefined behaviour) is typically strongly discouraged, particularly where safety is a factor, as in the MISRA C and C++ coding standards, for example.

Recently, structure-sensitive analysis [7] (hereon referred to as `cclyzer-ss`) presented a type-based flow-insensitive points-to analysis that enhances the precision of Andersen’s analysis [4]. `cclyzer-ss` lazily infers the types of heap objects by leveraging LLVM type casts to filter out spurious field derivations (i.e., field derivations introduced by static imprecision that can never happen during runtime).

1.4 Our solution

Inspired by TBAA and `cclyzer-ss`, the scope of this work is to produce a more precise flow-sensitive heap model for C and C++ programs which follow the strict aliasing rules. We propose `TYPECLONE`, a flow- (and field-) sensitive analysis with a new type-based heap abstraction which yields better precision than a traditional flow-sensitive analysis. We aim to perform lazy heap cloning by incorporating lightweight type information within standard flow-sensitive pointer analysis. Rather than performing heap cloning per calling context for each allocation, for untyped memory object o allocated at program point ℓ , we lazily clone to create typed object o_t at each of its type initialisation points ℓ' where the object o has a type t assumed. To maintain soundness, each clone o_t is back-propagated to any pointer that may have actually been accessing the concrete objects now represented by o_t through o .

Intuitively, the type initialisation point is treated as the real allocation site during our lazy heap cloning, thus distinguishing different sets of concrete objects where necessary in a lazy rather than eager manner. From the type initialisation point ℓ' forward, we only propagate the clone object o_t rather than the untyped object o . Not only are untyped objects prevented from propagating past type initialisation points like ℓ' , but any object of type t accessed by a pointer with incompatible element type t' will be strongly updated (i.e. killed, filtered) because such a points-to relation is impossible in a program which adheres to the strict aliasing rules and must be a result of static imprecision. This reduces the number of spurious objects in points-to sets during points-to resolution, especially when a killed object would have otherwise created spurious field sub-objects via field-sensitivity. This gives us a flow-sensitive analysis that is still scalable yet more precise than that which uses standard heap abstractions. We see our work more as an addition to flow-sensitive analysis rather than as competition to other heap cloning techniques like context-sensitivity. Our technique is neither strictly superior nor inferior to context-sensitive analysis and can work with context-sensitivity to achieve a more precise result. Our key contributions are summarised as follows:

- We present `TYPECLONE`, a flow-sensitive pointer analysis which can perform lazy heap cloning using types without context-sensitivity for C and C++ programs which do not violate the strict aliasing rules.
- We present new forms of strong updates, namely type-based semi-strong updates and type-based strong updates, which improve precision.
- We have implemented `TYPECLONE` and compared it to sparse flow-sensitive analysis. We have found that `TYPECLONE` can answer over 15% more alias queries with a no-alias result, on average, than `SPARSE`.

2 A motivating example

Figure 1a gives an example of a common heap allocation wrapper extracted from GNU Coreutils [18] and usage of that wrapper. This example aims to demonstrate the key idea of `TYPECLONE` and compare it with existing C and C++ points-to analyses: a recent flow-insensitive analysis (`cclyzer-ss`) and a flow-sensitive and context-insensitive analysis

<pre> 1 int main(void) { 2 int *p = (int *)xmalloc(4); 3 *p = 1; 4 float *q = (float *)xmalloc(4); 5 *q = 1.0; 6 // Alias(p, q)? 7 } 8 9 void *xmalloc(size_t n) { 10 void *x = malloc(n); 11 if (!x && n != 0) xalloc_die(); 12 return x; 13 }</pre>	<table border="1" style="border-collapse: collapse; width: 100%; text-align: left;"> <thead> <tr> <th style="border-bottom: 1px solid black;">Analysis</th> <th style="border-bottom: 1px solid black;">Points-to relations</th> <th style="border-bottom: 1px solid black;">ℓ_6: Alias(p,q)</th> </tr> </thead> <tbody> <tr> <td style="border-bottom: 1px solid black;">cclzyzer-ss</td> <td style="border-bottom: 1px solid black;">$\{o, o_{\text{int}}, o_{\text{float}}\} \in pt(\mathbf{p})$</td> <td style="border-bottom: 1px solid black;">TRUE</td> </tr> <tr> <td style="border-bottom: 1px solid black;">SPARSE</td> <td style="border-bottom: 1px solid black;">$\{o\} \in pt(\ell_{2-}, \mathbf{p})$</td> <td style="border-bottom: 1px solid black;">TRUE</td> </tr> <tr> <td style="border-bottom: 1px solid black;">TYPECLONE</td> <td style="border-bottom: 1px solid black;">$\{o_{\text{int}}\} \in pt(\ell_{3-}, \mathbf{p})$</td> <td style="border-bottom: 1px solid black;">FALSE</td> </tr> <tr> <td></td> <td>$\{o, o_{\text{int}}, o_{\text{float}}\} \in pt(\ell_4, \mathbf{q})$</td> <td></td> </tr> <tr> <td></td> <td>$\{o_{\text{float}}\} \in pt(\ell_{5-}, \mathbf{q})$</td> <td></td> </tr> </tbody> </table>	Analysis	Points-to relations	ℓ_6 : Alias(p,q)	cclzyzer-ss	$\{o, o_{\text{int}}, o_{\text{float}}\} \in pt(\mathbf{p})$	TRUE	SPARSE	$\{o\} \in pt(\ell_{2-}, \mathbf{p})$	TRUE	TYPECLONE	$\{o_{\text{int}}\} \in pt(\ell_{3-}, \mathbf{p})$	FALSE		$\{o, o_{\text{int}}, o_{\text{float}}\} \in pt(\ell_4, \mathbf{q})$			$\{o_{\text{float}}\} \in pt(\ell_{5-}, \mathbf{q})$	
Analysis	Points-to relations	ℓ_6 : Alias(p,q)																	
cclzyzer-ss	$\{o, o_{\text{int}}, o_{\text{float}}\} \in pt(\mathbf{p})$	TRUE																	
SPARSE	$\{o\} \in pt(\ell_{2-}, \mathbf{p})$	TRUE																	
TYPECLONE	$\{o_{\text{int}}\} \in pt(\ell_{3-}, \mathbf{p})$	FALSE																	
	$\{o, o_{\text{int}}, o_{\text{float}}\} \in pt(\ell_4, \mathbf{q})$																		
	$\{o_{\text{float}}\} \in pt(\ell_{5-}, \mathbf{q})$																		

(a) Usage of GNU Coreutils malloc wrapper. (b) Points-to relations.

■ **Figure 1** A motivating example.

(SPARSE). Points-to results are shown in Figure 1b. Flow-sensitive analyses maintain different points-to information for the same pointer at different program points, so pt takes an extra argument indicating the program point ($\ell-$ means ℓ onwards). The results show that TYPECLONE is more precise than cclzyzer-ss and SPARSE when performing an alias query on \mathbf{p} and \mathbf{q} at line 6 and maintains only one object at lines 3 and 5 onward.

cclzyzer-ss performs heap cloning at cast instructions, i.e., $p = (t) q$, such that the untyped object o that q points to is cloned to create a new object o_t with type t . The o_t is then propagated back to the allocation site of o in order to maintain soundness since some preceding program points may have become aliases of q , through points-to relations with the untyped object, before the clone was created. cclzyzer-ss’s main goal is to enable more precise field-sensitivity through these typed heap objects. Given a field constraint, i.e., $p = \&q \rightarrow f$, a new field is derived only when (1) q points to a typed object o_t , and (2) the type of q is $t*$. This prevents the generation of spurious field objects. cclzyzer-ss can only improve precision by preventing the generation of spurious field objects since every clone is back-propagated to its original allocation site which causes every object which originally pointed to the untyped object from that allocation site to point to every typed clone. For example, the clone object o_{int} , created at line 2, is back-propagated to line 10, which makes pointer \mathbf{q} , soundly but imprecisely, also point to o_{int} at every program point.

SPARSE [20] performs flow-sensitive points-to analysis using allocation-site-based heap modeling. Therefore, the program only has one heap object o which is then pointed to by both pointers, \mathbf{p} and \mathbf{q} , at line 6.

TYPECLONE clones heap objects lazily only at the type initialisation point in a flow-sensitive manner. A type initialisation point is any program point in which an object must be of a certain type (or one of a set of types) for the program to be legal (i.e., not exhibit undefined behaviour). In this example, the pointer dereferences at lines 3 and 5 must be referring to objects of type `int` and `float`, respectively, for the program to avoid undefined behaviour. This is in contrast to cclzyzer-ss which clones at cast instructions eagerly. The cloned objects are then propagated back to their allocation sites for the same reason that cclzyzer-ss does and so \mathbf{x} at line 10 would then point to the two cloned objects (as well as the original untyped object). When the clone objects o_{int} and o_{float} are re-propagated to line 3 following the control-flow, o_{float} will be filtered by TYPECLONE’s type-based strong updates since our analysis only expects an `int`-typed object (o_{int}) to be accessed by the

■ **Table 1** Domains and LLVM-like instructions used by our pointer analysis.

Analysis domains			LLVM-like instruction set	
l	$\in \mathcal{L}$	instruction labels	STK/GLOB	$p = \&o$
t, \bullet	$\in \mathcal{T}$	types	HEAP	$p = \text{malloc}_o$
k	$\in \mathcal{C}$	constants	PHI	$p = \phi(q, r)$
i	$\in \mathcal{S}$	stack virtual registers	FIELD	$p = \&q \rightarrow f_k$
g	$\in \mathcal{G}$	global variables	CAST	$p = (t) q$
$p, q, r, x, y \in \mathcal{P} = \mathcal{S} \cup \mathcal{G}$		top-level variables	LOAD	$p = *q$
\dot{o}	$\in \mathcal{O}$	abstract objects	STORE	$*p = q$
$o.f_k$	$\in \mathcal{F}$	abstract field objects	CALL	$p = q(r_1, \dots, r_n)$
a, o	$\in \mathcal{A} = \mathcal{O} \cup \mathcal{F}$	address-taken variables	FUNENTRY	$fun(r_1, \dots, r_n)$
v	$\in \mathcal{V} = \mathcal{P} \cup \mathcal{A}$	program variables	FUNEXIT	$ret_{fun} p$

dereference $*p$. Similarly, a type-based strong update is also performed at line 5 to kill the incompatible typed object o_{int} . Thus, TYPECLONE is able to more precisely answer the alias query at line 6 than both SPARSE and cclzyzer-ss.

3 Program representation and type model

This section describes the program representation, our type model (based on the C and C++ standards), and the value-flow representation used for our flow-sensitive analysis.

3.1 Program representation

Like [7, 20, 30, 39], we perform our pointer analysis on top of the LLVM IR of a program. The instructions relevant to our analysis and the domains are given in Table 1. The set of all variables \mathcal{V} is separated into two subsets: $\mathcal{A} = \mathcal{O} \cup \mathcal{F}$ which contains all possible abstract objects and their fields (i.e., *address-taken variables* of a pointer), and \mathcal{P} which contains *top-level variables*, including stack virtual registers (symbols starting with % in LLVM) and global variables (symbols starting with @). Top-level variables in \mathcal{P} are explicit and directly accessed, and address-taken variables in \mathcal{A} are implicit and indirectly accessed at LLVM LOAD or STORE instructions through top-level variables. Since our analysis is type-based, we require types: $t \in \mathcal{T}$. $\bullet \in \mathcal{T}$ is the undefined type which is necessary because heap objects can be untyped before being initialised.

After SSA (static single assignment) conversion, given that $p, q, r_1, \dots, r_n \in \mathcal{P}$ and $o \in \mathcal{A}$, a program is represented by ten types of instructions: (1) eight types of instructions which appear in the body of a function: $p = \&o$ (allocates memory for a stack or global object), $p = \text{malloc}_o$ (allocates memory for a heap object), $p = \phi(q, r)$ (selects the value of a variable at the joint point of branching control-flow), $p = \&q \rightarrow f_k$ (retrieves a pointer pointing to the field of a struct object), $p = *q$ (reads the value of an object), $*p = q$ (writes the value of an object), $p = (t) q$ (casts a pointer to type t), and $p = q(r_1, \dots, r_n)$ (calls function q with arguments r_1, \dots, r_n), and, (2) a FUNENTRY instruction $fun(r_1, \dots, r_n)$ containing the parameters of fun , and a FUNEXIT instruction $ret_{fun} p$ representing the unique return of fun . LLVM pass `UnifyFunctionExitNodes` is executed before pointer analysis to ensure that every function has only one FUNEXIT instruction. Top-level variables are put directly in SSA form, while address-taken variables are only accessed indirectly via LOAD or STORE instructions. Parameter passing and returning are treated as COPYs.

<pre> 1 p = &a; 2 a = &b; 3 4 5 q = malloc(...); 6 *q = &c; 7 8 9 *p = *q; 10 </pre> <p>(a) C code fragment.</p>	<pre> 1 p = &a; 2 x₁ = &b; 3 *p = x₁; 4 5 q = malloc(...); 6 x₂ = &c; 7 *q = x₂; 8 9 x₃ = *q; 10 *p = x₃; </pre> <p>(b) Corresponding LLVM IR.</p>
---	---

Figure 2 C code and its corresponding LLVM IR.

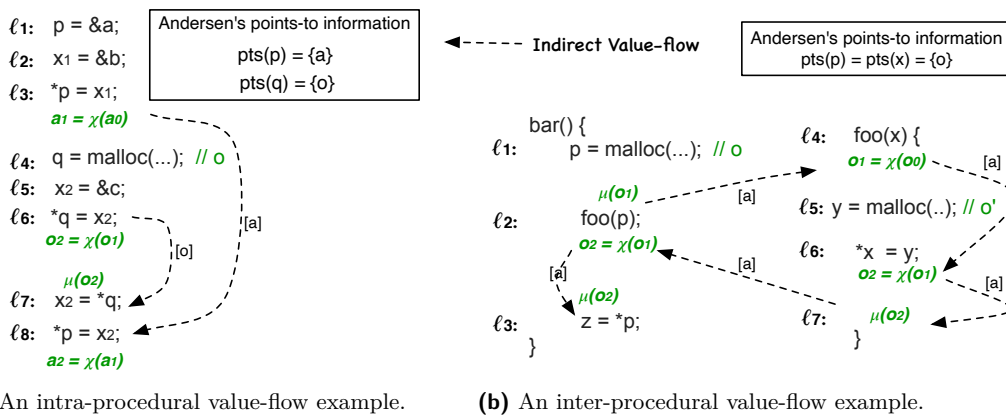


Figure 3 Intra-procedural and inter-procedural value-flow graph examples.

C++ object allocation, like $S *x = \text{new } S()$, is generally translated into two instructions: an allocation instruction, like $x = \text{malloc}_o$, which returns an untyped object, and a call to a constructor, like $S(x)$, which calls S 's constructor to initialise the newly allocated object. For pointer arithmetic, as in $q = p + j$, if p points to an object o , we conservatively assume that q can point anywhere within o . This is based on the assumption that pointer arithmetic cannot cross the boundary of an object. A pointer pointing to an object produced through an integer cast aliases every other pointer (i.e. it points to every object). We treat arrays monolithically such that accessing any element of an array is the same as accessing the entire array object meaning our analysis is array-insensitive. This is an orthogonal dimension of precision.

Given $p, q, x_1, x_2, x_3 \in \mathcal{P}$ and $a, b, c \in \mathcal{A}$, Figure 2 shows a code fragment and its corresponding LLVM partial SSA form. Note that address-taken variable a can only be indirectly accessed by introducing a top level pointer, for example through x_1 in store $*p = x_1$. Complex statements like $*q = \&c$ and $*p = *q$ are decomposed into basic instructions by introducing top-level pointers like x_2 and x_3 .

3.2 Value-flow representation for flow-sensitive analysis

Unlike a flow-insensitive analysis which ignores program execution order, a flow-sensitive analysis accounts for the program's control flow. Traditional flow-sensitive points-to analysis computes and maintains data-flow facts (points-to relations) at each program point. These

data-flow facts are propagated along the program’s inter-procedural control-flow graph (ICFG) until a fixed point is reached [12, 30]. However, in time and space, computing and propagating the points-to information on the ICFG is costly.

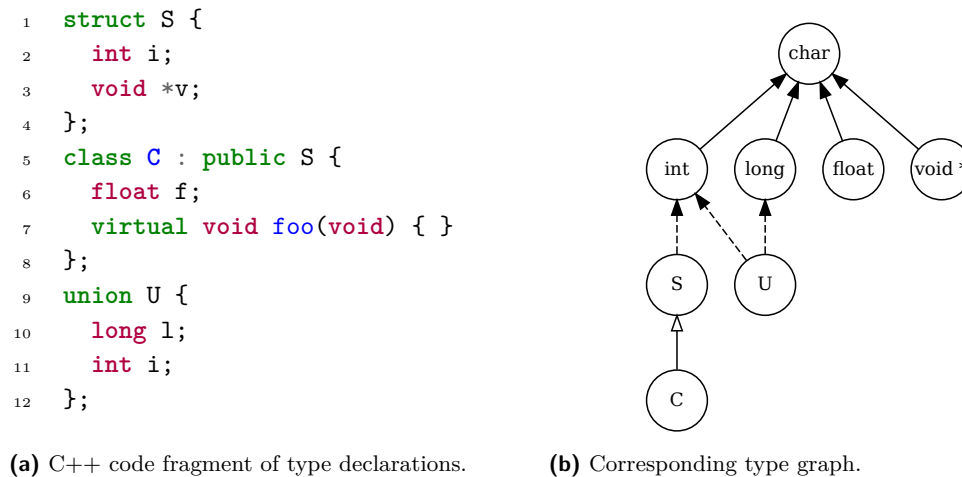
To accelerate performance, the analysis is run on top of a sparse value-flow graph (VFG) instead of the ICFG [20, 33, 36, 39]. Intuitively, the VFG is a sparse def-use graph where each node represents a program statement and edges represent a def-use chain (i.e. value-flow) of a program variable. The notion of sparsity comes from the omission of statements irrelevant to the value-flow of a variable such that an edge acts as a “jump” directly from the definition of a variable to its use thus reducing redundant points-to propagation along the control-flow.

The value-flow of a top-level variable (which has a unique definition in the partial SSA form) is directly available without pointer analysis. Such value-flows are called *direct value-flows*. A directed edge from node x to node y represents a value-flow relation: a variable defined at node x is used at node y . On the other hand, value-flows of address-taken variables (which are not in partial SSA form) are obtained by building the memory SSA form following [11, 20] because their uses, at loads for example, could be defined indirectly at multiple stores

Figure 3 shows an intra- and an inter-procedural example based on the code in Figure 2. For both, a fast and imprecise flow-insensitive Andersen’s analysis [4] is used to annotate indirect memory accesses at program points like loads, stores, and callsites. The results of that Andersen’s analysis are shown in the two boxes in Figures 3a and 3b. Now, let us consider the intra-procedural example in Figure 3a specifically. Firstly, we annotate store instructions like $*p = x_1$ with a function $a = \chi(a)$ for each variable $a \in \mathcal{A}$ which may be pointed to by p . This represents a potential value-flow (i.e., def and use of a) at the store. If a can be strongly updated, then a receives the value on the right-hand side of the store (x_1) and the old contents in a are killed. Otherwise, a weak update takes place by adding the right-hand side to a ’s old contents [30]. Secondly, we annotate load instructions like $x_2 = *q$ with a function $\mu(o)$ for each variable $o \in \mathcal{A}$ that may be pointed to by q to represent a potential use of o at the load. Thirdly, we convert all address-taken variables into SSA form, treating each $\mu(o)$ as a use of o and each $o = \chi(o)$ as both a def and a use of o . Finally, we obtain the indirect value-flows for $o \in \mathcal{A}$: for a use of o , identified as o_n (where n represents the version), at load or store instruction ℓ , with its unique definition at store ℓ' , $\ell' \xrightarrow{o} \ell$ represents the potentially *indirect value-flow* of o from ℓ' to ℓ . This is exemplified by $\ell_3 \xrightarrow{a} \ell_8$ and $\ell_6 \xrightarrow{o} \ell_7$ in Figure 3a.

Figure 3b shows an inter-procedural value-flow example. In addition to what is done in the intra-procedural case, we compute the side-effects of a function call by applying a lightweight inter-procedural mod-ref analysis [41, §4.2.1]. A given callsite, ℓ , is annotated with $\mu(a)$, or $a = \chi(a)$, if a may be read, or modified, respectively, inside the callees of ℓ as discovered by Andersen’s pointer analysis. Additionally, appropriate χ and μ operators are also added to the FUNENTRY and FUNEXIT instructions of these callees in order to mimic parameter passing and returning of address-taken variables.

To handle read side-effects within a function, we add a μ call before appropriate callsites to represent potential uses of μ ’s argument. The corresponding FUNENTRY instruction is annotated with an appropriate χ call. For example, to represent potential uses of o in `foo` in Figure 3b, $\mu(o)$ is added before the callsite at ℓ_2 , and `foo`’s FUNENTRY instruction at ℓ_4 is annotated with $o = \chi(o)$ to receive the values of o passed from ℓ_2 . Similarly, for modification side-effects within a function, a call to χ is added after appropriate callsites to receive potentially modified values, and the corresponding functions’ FUNEXIT instructions are annotated with a μ call. In Figure 3b, $o = \chi(o)$ is added after the callsite at ℓ_2 to handle



■ **Figure 4** A type graph and the corresponding C++ code it was generated from. The open triangle arrow represents an inheritance relation, the dashed arrow represents a first-field relation, and the solid arrow represents relations derived from the strict aliasing rules of C and C++.

potential modification of o in `foo`, and `foo`'s `FUNEXIT` instruction at ℓ_7 is annotated with $\mu(o)$. Finally, as in the intra-procedural example, all variables in μ and χ are renamed and converted into memory SSA form when connecting their def-use relations.

3.3 Type model

The types of stack and global objects are static and are determined at their allocation sites whereas dynamically allocated heap objects (e.g., through `malloc` or C++'s `new`) are untyped. Primarily, the type of a heap object is manifested when accessed through the dereferencing of a pointer.

Before introducing the type model for our analysis, we first define the subtyping relations that may appear in a program. We use a type graph to represent subtyping relations between different types in a program. The type graph is a directed acyclic graph where each node represents a type in the target program and each edge from t' to t represents one of three subtyping relations: (1) t' inherits from t , (2) the first field of struct type t' has type t , and (3) subtyping relations derived from the strict aliasing rules of C and C++. All types appearing in the program are placed into the type graph, but we treat arrays and pointers as equivalent for this purpose and, in conformance with the strict aliasing rules, ignore signedness and qualifiers (e.g., `int` is deemed equivalent to `const unsigned int`).

Figure 4 illustrates the three subtyping relations on an example type graph (Figure 4b) generated from the code in Figure 4a. Class `C` inherits from struct `S` which has an `int` as its first field. Unions treat all their members as their first field, as is the case with `U` and its `int` and `long` members. Finally, all types which do not have any outgoing edges are connected to the `char` type in line with the strict aliasing rules which allow any object to be accessed by dereferencing a pointer to a `char`. Finally, we say that t' is a subtype of t , denoted as $t \prec t'$, if t is reachable from t' in the type graph.

Our analysis expects target programs to conform to our type model. We define the type model of our analysis with the following rules **R1–R5**, adopted from the provenance and strict aliasing rules of C18 [2] and C++17 [1].

24:10 Flow-Sensitive Type-Based Heap Cloning

- R1. An abstract heap object's initial type is undefined (\bullet) until a non-void type is potentially assigned.
- R2. Any pointer may point to any object regardless of its type (through pointer casting, for example). However, through a pointer of type t^* only objects whose type is t' , such that $t \prec t'$, may be read.
- R3. For pointer arithmetic $q = p + j$, q will either point within the object pointed to by p , or at one past the last element of the object if it is an array object.
- R4. An object may not access any of its virtual methods until it is passed to the corresponding constructor.
- R5. An object's type can be changed to a type that is not a transitive base (i.e., reuse).

R1–R4 are easy to understand. R5 describes object reuse in C and C++. An example use case is to reuse an already allocated object rather than freeing that object and performing a new allocation. This is commonly done using placement `new` in C++. Another use case would be custom allocators in C++ using a statically allocated pool of memory, e.g., given a static buffer (or allocated otherwise), `char buf[100]`, a placement `new` operation, `new(buf) T()`, would not allocate new memory, but would call `T`'s constructor with `buf` as the `this` argument to initialise the underlying object with type `T`. In C, placement `new` is unavailable. Outside well know patterns like pool allocators, object reuse is an uncommon feature that is error-prone since it may make pointers illegal to dereference, unless strict conditions are met. This is similar to introducing dangling pointers through deallocation except that deallocation is usually more explicit (e.g., through the presence of `free` and `delete`) and more commonly understood by the programmer than the possible ways of performing reuse. We discuss object reuse in more detail in Section 4.2.

4 TypeClone approach

We present a base analysis in Section 4.1 that will achieve our goals of typing heap objects and performing flow-sensitive heap cloning without context-sensitivity. The base analysis assumes no direct object reuse. We then extend the base analysis to support direct object reuse in Section 4.2.

4.1 Base analysis

This section introduces our base analysis. Central to our analysis is typing the usually untyped abstract objects. We use the notation o_t to indicate that the type of object o is t and we use o without the type subscript when the type is irrelevant. For stack and global objects, the type is assigned at allocation, whereas for heap objects, the type is undefined (denoted as \bullet) at allocation. According to our type model in Section 3.3, pointers with element type t can only read from objects whose underlying type t' satisfies $t \prec t'$. Therefore, such pointer accesses are an indication of the type of an object and we can use this information for heap cloning. Writes to an object are another indication of the type of an object: an object written to through a pointer with element type t is assigned type t . Figure 5 presents the inference rules for the base analysis and an explanation of each of the rules follows.

4.1.1 Memory allocation ([HEAP] [STACK/GLOBAL])

The [HEAP] and [STACK/GLOBAL] rules handle the allocation of heap, stack and global objects. Allocation is handled as in standard flow-sensitive pointer analysis except that we associate a type with the newly allocated objects. At allocation time, the type of a heap

$$\begin{array}{c}
\text{[STACK/GLOBAL]} \\
\frac{\ell : p = \&o \quad t = T(p)}{\boxed{O_{\tilde{t}}} \in pt(\ell, p)} \\
\text{[LOAD]} \\
\frac{\ell : p = *q \quad \ell'' \xrightarrow{q} \ell \quad \ell' \xrightarrow{o} \ell \quad o_t \in pt(\ell'', q) \quad o' = \mathbf{init}(T(q), o_t)}{pt(\ell', o') \subseteq pt(\ell, p)} \\
\mathbf{init}(t, o_{t'}) = \begin{cases} \boxed{O_{\tilde{t}}} & \text{if } t' \equiv \bullet \text{ [INITIALISE]} \\ o_{t'} & \text{if } \tilde{t} \prec t' \text{ [TBWU]} \\ \boxed{O_{\tilde{t}}} & \text{if } t' \prec \tilde{t} \wedge t' \not\equiv \tilde{t} \wedge h(o_{t'}) \text{ [TBSSU]} \\ \emptyset & \text{otherwise [TBSU]} \end{cases} \\
\text{[SU/WU]} \\
\frac{\ell : *p = _ \quad \ell' \xrightarrow{o} \ell \quad o \in \mathcal{A} \setminus \mathbf{kill}(\ell, p)}{pt(\ell', o) \subseteq pt(\ell, o)} \quad \mathbf{kill}(\ell, p) = \begin{cases} \{o'\} & \text{if } pt(\ell, p) \equiv \{o'\} \wedge o' \in \text{singletons} \\ \mathcal{A} & \text{if } pt(\ell, p) \equiv \emptyset \\ \emptyset & \text{otherwise} \end{cases} \\
\text{[FIELD]} \\
\frac{\ell : p = \&q \rightarrow f_k \quad \ell' \xrightarrow{q} \ell \quad o \in pt(\ell', q) \quad o' = \mathbf{init}(T(q), o) \quad t = \text{type of } t::f_k}{\boxed{(o'.f_k)_t} \in pt(\ell, p)} \quad \text{[FF-NOT-IN-PT]} \quad \frac{o \sim o' \quad o \in pt(\ell, v)}{o' \in pt(\ell, v)} \quad \text{[FF-EQ-PT]} \quad \frac{o' \sim o'' \quad o \in pt(\ell, o')}{o \in pt(\ell, o'')} \\
\text{[PHI]} \\
\frac{\ell : p = \phi(q, r) \quad \ell' \xrightarrow{q} \ell \quad \ell'' \xrightarrow{r} \ell}{pt(\ell', q) \cup pt(\ell'', r) \subseteq pt(\ell, p)} \quad \text{[CAST]} \\
\frac{\ell : p = (t) \quad q \quad \ell' \xrightarrow{q} \ell \quad o \in pt(\ell', q)}{o \in pt(\ell, p)} \\
\text{[CALL]} \\
\frac{\ell : _ = q(\dots, r, \dots) \quad \mu(o) \quad \ell' : \mathit{fun}(\dots, r', \dots) \quad o = \chi(o) \quad o_{fun} \in pt(\ell'', q) \quad \ell'' \xrightarrow{q} \ell \quad \ell^* \xrightarrow{r'} \ell \quad \ell^* \xrightarrow{o} \ell}{pt(\ell^*, r) \subseteq pt(\ell', r') \quad pt(\ell^*, o) \subseteq pt(\ell', o)} \\
\text{[RET]} \\
\frac{\ell : p = q(\dots) \quad o = \chi(o) \quad \ell' : \mathit{ret}_{fun} p' \quad \mu(o) \quad o_{fun} \in pt(\ell'', q) \quad \ell'' \xrightarrow{q} \ell \quad \ell^* \xrightarrow{p'} \ell \quad \ell^* \xrightarrow{o} \ell}{pt(\ell^*, p') \subseteq pt(\ell, p) \quad pt(\ell^*, o) \subseteq pt(\ell, o)} \\
T(v) : \mathcal{V} \mapsto \mathcal{T} \quad v\text{'s type.} \quad \tilde{t} : \mathcal{T} \mapsto \mathcal{T} \quad \text{The element (pointee) type of } t. \\
pt(\ell, v) : \mathcal{L} \times \mathcal{V} \mapsto 2^{\mathcal{A}} \quad v\text{'s points-to set after } \ell. \quad t \prec t' : \mathcal{T} \times \mathcal{T} \quad t' \text{ is a transitive subtype of } t. \\
\ell \xrightarrow{v} \ell' : \mathcal{L} \times \mathcal{V} \times \mathcal{L} \quad v\text{'s value flow.} \quad \bullet : \mathcal{T} \quad \text{The undefined type.} \\
h(o) : \mathcal{A} \mapsto \text{Bool} \quad o \text{ is a heap object.}
\end{array}$$

■ **Figure 5** Inference rules for the base analysis. We use a $\boxed{\text{box}}$ to indicate that a new cloned object is created if it does not already exist.

object is unknown and will be determined later through usage of the object. Thus an untyped object o_\bullet is propagated. The types of stack and global objects, however, are known, so a type (t in o_t) is immediately assigned.

4.1.2 Direct and indirect propagation ([PHI] [CAST] [CALL] [RET])

Rules [PHI] and [CAST] act as copies, performing trivial direct propagation. Both simply propagate the points-to information of the pointer on the right hand side of an assignment to the pointer on the left hand side. In the case of rule [PHI], the points-to sets of q and r are added to that of p . In the case of rule [CAST], the points-to set of q is added to that of p as a CAST instruction acts like a copy. Despite the type-based nature of our analysis type casting has no effect on actual points-to relations since any pointer can point to any object except when that pointer is used in certain ways (recall **R2** of our type model).

Direct inter-procedural points-to propagation is done via the [CALL] and [RET] rules. Function targets are resolved on-the-fly during points-to analysis to more precisely discover callee functions at indirect callsites. Points-to values from the actual arguments at the callsite are then propagated to the corresponding formal parameters of each callee. The points-to information of an address-taken variable o is propagated via indirect value-flows (Section 3.2) with the χ and μ annotations shown in the two rules.

Indirect propagation is the propagation of address-taken variables (objects in \mathcal{A}) in the VFG. Indirect edges are labelled with address-taken variables, as determined by the pre-analysis, which are then propagated along those edges. Since the Andersen's analysis used to construct the VFG has no notion of typed objects, the edges are labelled with objects according to the allocation-site-based heap model (i.e. the original untyped objects). To remedy this, on the fly, when an indirect edge is labelled o , we propagate points-to information for all clones of o defined at the source of the indirect edge through the indirect propagation of o . For example, the propagation of the points-to information of both o_t and $o_{t'}$ from ℓ and ℓ' would be through the indirect edge labelled with o , i.e. $\ell \xrightarrow{o} \ell'$. Another way to resolve this is to augment the pre-analysis with type-based heap cloning and then indirect edges would be labelled with the appropriate typed objects. We forego this method for brevity and generality.

4.1.3 Loads and stores ([LOAD] [STORE] [SU/WU])

LOAD and STORE instructions, through the [LOAD] and [STORE] rules, are handled in the same way as in a standard sparse flow-sensitive analysis [20] except that object initialisation (via **init**, described in Sections 4.1.4 and 4.1.5) is performed on the objects pointed to by the dereferenced pointers. The return value of the **init** function is then operated on rather than the object in the dereferenced pointer's points-to set (which was passed in to **init**).

The [SU/WU] rule performs standard singleton-based strong and weak updates [20, 30] for object o , pointed to by p , at STORE instruction $*p = q$. A weak update merges the points-to set of o with that of q and propagates that result onward. When o is a singleton, a strong update is performed. Strong updates discard o 's old pointees, making its points-to set equivalent to that of q [30]. Singleton-based strong updates cannot take place upon local variables within recursion, arrays (treated monolithically), and heap objects. In addition to these singleton-based strong updates, TYPECLONE performs type-based semi-strong updates (Section 4.1.5.2) and type-based strong updates (Section 4.1.5.3) by leveraging our typing of abstract objects.

```

1  class S { ... };
2  class T : public S { int f; };
3  S *smalloc(size_t size) {
4      return (S*)malloc(size);
5  }
6  int main(void) {
7      T *t = (T*)smalloc(sizeof(T));
8      t->f = ...;
9  }

```

■ **Figure 6** An example where object cloning would be performed by `clyzer-ss` but not by `TYPECLONE`.

4.1.4 Object cloning

Object initialisation occurs whenever pointer access to an object makes an assumption regarding that object's type. The `LOAD`, `STORE`, and `FIELD` instructions make assumptions about the type of the object being accessed. The `init` function used by rules `[LOAD]`, `[STORE]`, and `[FIELD]` handles object cloning through four different cases: `[INITIALISE]`, `[TBWU]`, `[TBSSU]`, and `[TBSU]`. It takes two arguments: the type t of the pointer pointing to the object of interest and the object o_v being accessed by the pointer. `init` may produce new objects if they do not already exist (i.e., it may clone objects). The potential to create a new object is denoted by a `box` and the $\tilde{}$ operator, as in \tilde{t} , returns the element type of a pointer such that it would return `int *` when applied to `int **`, for example.

4.1.4.1 Object initialisation (`[INITIALISE]`)

In the `[INITIALISE]` case, an untyped object ($t' = \bullet$) is accessed by a pointer of type t . `TYPECLONE` will initialise the type of the heap object to be \tilde{t} based on the assumption that the underlying type of the object is of type \tilde{t} or a subtype of \tilde{t} . We can then propagate the \tilde{t} typed object and stop propagating the untyped object thus differentiating it from objects of different types originating at the same allocation site. In C and C++, for example, code snippet `int *i = (int *)malloc(4); *i = 1;` makes an assumption about the type of the object returned by `malloc`—that it is of type `int` or a subtype of `int`—because pointer `i` cannot be accessing an object of any other type per our model.

Our approach to object cloning differs from that of `clyzer-ss` in that it is less eager. When a pointer is cast to a pointer to another type, but never dereferenced as that pointer type, our approach would not perform object cloning whereas `clyzer-ss` would. For example, consider Figure 6 where `T` is a derivative type of `S`. Despite a pointer to the allocated object o being cast to type `S *`, it is never dereferenced as such, and so we do not need to clone to create o_S .

4.1.4.2 Back propagation (`[BACK-PROPAGATE]`)

In the `[INITIALISE]` case, a new object is created and solely returned by `init`, causing the caller to stop propagating the original object, and to only propagate the clone. This ignores aliases made before the object was initialised and assigned a type. Figure 7 exemplifies this where pointer `a` is assigned pointer `i` before the pointed-to heap object is initialised at line 3. At lines 1 and 2, `i` would point to the untyped object o_\bullet , `a` would also point to o_\bullet at lines 2 and 3, and `i` would correctly point (only) to the typed object o_{int} at line 3 but would simultaneously share an incorrect no-alias relation with `a`.

```

1  int *i = (int *)malloc(...);
2  void *a = i;
3  *i = 1;

```

■ **Figure 7** An example of an alias being made before initialisation occurs.

The [BACK-PROPAGATE] rule ensures that pre-initialisation aliases, like `a` in Figure 7, also point to the clone by back-propagating the newly created clone to the original object’s allocation site, like [6, 7]. This is a cause of imprecision, since more than just pre-initialisation aliases will now point to the clone. Some precision is then recouped by subsequent typed-based strong and semi-strong updates.

4.1.5 Type-based weak, semi-strong, and strong updates

The `init` function acts upon types, hence we say it performs type-based updates. These updates are divided into type-based weak updates, type-based semi-strong updates, and type-based strong updates.

4.1.5.1 Type-based weak updates ([TBWU])

The [TBWU] case represents the basic case, *type-based weak updates*, or TBWUs. In this case, either the object’s type exactly matches the pointer’s element type ($\tilde{t} \equiv t'$), or the object’s type is a derived type of the pointer’s element type (an upcast took place). Both situations are covered by the statement $\tilde{t} \prec t'$. Since this access asserts the legality of such a pointer accessing such an object and makes no new assumptions about the object’s type, it is simply propagated onward like a standard flow-sensitive analysis would and no cloning occurs.

4.1.5.2 Type-based semi-strong updates ([TBSSU])

The first case, [INITIALISE], results in a *type-based semi-strong update*, or TBSSU. It is type-based since the mechanism by which it occurs relies on type information, and it is semi-strong in that it kills one object and replaces it with another.

The [TBSSU] case, which models access after a downcast occurs, also results in a type-based semi-strong update. TYPECLONE assumes that any pointer access resulting from a downcast (i.e., $p = (t) q$ where $T(p) \prec T(q)$) is legal since we consider all input programs to conform to the strict aliasing rules of C and C++ (as implied by our type model). Regardless of whether an analysis accepts illegal programs or not, this is also the more conservative way of handling access after a downcast (with respect to soundness). From another point of view, we cannot know if the original type we assigned is the actual type of the object, and that the original type initialisation was actually an access through an upcast. We test that $o_{t'}$ is a heap object ($h(o_{t'})$) because non-heap objects have a declared type and cannot be changed (until we discuss reuse in Section 4.2).

Like the [INITIALISATION] case, we create a new object of type \tilde{t} since an assumption about the type of the object is being made (that its real type is \tilde{t} or a derivative of \tilde{t} , both of which being derived types of t'). It is tempting to change the object’s type instead of cloning but this can cause unsoundness as the abstract object $o_{t'}$ may have been representing both concrete objects of type t' and concrete objects of type \tilde{t} or other derivatives.

Handling downcasts explicitly gives a more accurate representation of an object’s type which is necessary for virtual method resolution, and allows for better strong updates. To illustrate the latter, consider the C code in Figure 8, which implements a rudimentary form of

```

1  typedef struct { int i; } S;
2  typedef struct { struct S s; long l; } T1;
3  typedef struct { struct S s; float f; } T2;
4  void *smalloc(size_t size) {
5      S *base = (S *)malloc(size);
6      base->i = 1;
7      return base;
8  }
9  int main(void) {
10     T1 *p = (T1 *)smalloc(sizeof(T1));
11     p->l = 2;
12     T2 *q = (T2 *)tmalloc(sizeof(T2));
13     q->f = 3.0;
14 }

```

■ **Figure 8** An example of an abstract object being initialised as two different “derivative” types, T1 and T2, after it has been initialised as the “base” type S.

inheritance. Within wrapper function `smalloc`, all allocated objects are initialised to “base” type S. Callers of `smalloc` can then access the returned object as a “derivative” type T1 or T2. If the programmer allocates the correct size, then this is legal since $S \prec T1/T2$. Since we see this as a downcast, the initialisation at line 11, for pointer `p`, would create a new object with type T1, stop propagating the object of type S, and back-propagate the new object. This would similarly occur at line 13 for pointer `q`. `p` and `q` at lines 11 and 13 would then *not* alias since they would perform a type-based strong update (discussed in the following section) on the back-propagated object which does not match their type, and neither would point to the S typed object any longer from the type-based semi-strong update to an object of type T1/T2. In essence, we have split an abstract object into more abstract objects, each of which representing a smaller set of concrete objects than the original abstract object.

4.1.5.3 Type-based strong updates ([TBSU])

When an object is typed and there is no relation between the object’s type and the pointer’s element type, we know that the pointer is pointing to an object which would be impossible during execution (i.e., a spurious object). A conforming program cannot, for example, read an object through an unrelated pointer. In the [TBSU] case, we return nothing, which is, in effect, a strong update; the pointer will not regard the killed object as in its points-to set. A *type-based strong update* (TBSU) differs from a typical strong update in that it applies to any potential initialisation points, uses type information to perform it, and can occur to all forms of abstract objects.

4.1.6 Field-sensitivity ([FIELD] [FF-NOT-IN-PT] [FF-EQ-PT])

Taking the address of a field of an aggregate object is handled by the [FIELD] rule. The [FIELD] rule is the same as that in a standard flow-sensitive analysis, except that (1) initialisation is performed on the objects which `q` points to since assumptions about the aggregate objects in question are being made, and (2) the type of the new field object is assigned by looking up the aggregate type of the object having its field taken. The [TBSU] case occurring on pointees of `q` has a similar effect to the filtering that `ccllyzer-ss` performs where no field object is created for a spurious aggregate object [6, 7].

24:16 Flow-Sensitive Type-Based Heap Cloning

```
1  int s;  
2  int *i = &s;  
3  *i = 1;  
4  float *f = new(i) float{2.0};  
5  *f = 2.0;  
6  // It is now undefined behaviour to load i.
```

■ **Figure 9** An example of object reuse.

Though not represented in the rules (for simplicity), rather than further deriving a field object from a field object, we add the field index to the existing field object. For example, rather than deriving field object $o.f_k.f_j$, we would derive $o.f_{k+j}$. Only dealing with field objects derived from aggregate objects makes reasoning about the analysis easier.

Struct objects in C and standard-layout objects in C++ share the same memory address as that of their first field. Our analysis must ensure there is an equivalence between the points-to sets of such objects and their first field. A non-standard-layout object in C++ does not have to alias its first programmer-defined field in C++. This is often the case in practice, as in Clang and GCC, due to the implementation placing a virtual table pointer at the start of some objects for example. However, for the purpose of a pointer analysis, the first field of an object is tracked, regardless of whether it is programmer-defined or not, so this “first-field aliasing” needs to be applied to all struct and class objects.

We follow [6] where changes involving an object or the first field of an object trigger changes in the other through rules [FF-NOT-IN-PT] and [FF-EQ-PT]. They both use the first-field alias relation, defined as follows, similar to [6],

► **Definition 1.** *First-field alias relation.* The first-field alias relation is defined as the equivalence relation $\sim : \mathcal{A} \times \mathcal{A}$ such that $o, o' \in \mathcal{A}$, $o \sim o'$ if and only if:

$$o' = o.f_0$$

The [FF-NOT-IN-PT] rule ensures that when an object o belongs to some points-to set, then so does its first field $o.f_0$, if it exists, and vice versa. The [FF-EQ-PT] rule ensures that when an object o is in the points-to set of an object o' , then it is also in the points-to set of its first-field $o'.f_0$, and vice versa. This keeps both points-to sets equivalent.

4.2 Object reuse

The base analysis presents a simple overview of our approach. This section extends our base analysis by considering object reuse, a special language feature in C and C++, to make the analysis handle programs making use of this feature. In C, writing to a heap object through a pointer of type t^* changes that object’s type to type t ; reads through a pointer of type t^* are now legal, and reads through pointers of type t'^* where $t' \neq t$ result in undefined behaviour. This would be only permitted if t fits in the space allocated for the object. Furthermore, in C, the type of an object (referred to as the “effective type” in the C standard) can be changed through functions `memcpy` and `memmove` or by being copied as a character array, thus not requiring a store through a pointer to the new type. Reuse is also possible in C++ with the addition that this may be achieved through placement `new` and that placement `new` can be used on stack and global objects. Though necessary to achieve better soundness, we have excluded reuse from the base analysis as it introduces a


```

1  char *pool;
2  void *palloc(size_t s) {
3      // Find index appropriate for s...
4      return pool + n;
5  }
6  void *pfree(char *m) {
7      // Return m to pool...
8  }
9  int main(void) {
10     pool = malloc(512);
11     int *i = palloc(sizeof(int));
12     *i = 1;
13     pfree(i);
14     float *f = palloc(sizeof(float));
15     *f = 2.0;
16 }

```

■ **Figure 10** An example of pool allocator.

performance penalty, slight imprecision, is not often used in many C/C++ programs, and can be error-prone outside common patterns like pool allocators (which our analysis can handle when the pool is from the heap). An example of reuse is shown in Figure 9.

The form of reuse shown in Figure 9 cannot generally be soundly handled by our base analysis. A pool allocator, like in Figure 10, on the other hand, can be handled soundly by our base analysis, despite it relying on reuse. In Figure 10 the memory given to `f` can be the same as that given to `i`. The difference from Figure 9 is that the memory object is assigned from the main pool, similar to how an object may flow from any allocation wrapper (around `malloc`). The untyped object from the pool would then be initialised. The base analysis cannot handle pool allocators where the pool is a stack or global object since such an object would not be untyped (and thus cannot be initialised).

The incompatibility between reuse and the base analysis stems from the `init` function. Typically, reuse would not be possible through a `LOAD`, nor a `FIELD` not being stored to. However, if we limit reuse to stores, the analysis may be unsound since the case of changing the type of an object through `memcpy/memmove` or copying as a `char` array would need to be specially handled. Thus, we allow for reuse at `LOAD`, `STORE`, and `FIELD` instructions. The `init` functions is changed to that in Figure 11.

The new case in the `init` function, `[REUSE]`, checks for an incompatibility between the object's type and the pointer's element type. If the types are incompatible, a clone is returned with the new type, that is, the object is being reused with the new type. This is a semi-strong update similar to the `[TBSSU]` and `[INITIALISATION]` cases since the previous object is no longer propagated and another is instead. We also remove the $h(o_v)$ condition from the `[TBSSU]` case to conservatively implement C++'s allowance of reuse of stack and global objects, as a stack or global object may be accessed through a pointer to a derived type (i.e., through downcasting) with placement `new`. It is not necessary to specially handle placement `new` since the object will eventually be written to as the new type, or read from as such. The `[TBSU]` will no longer be triggered as all cases are now covered with the introduction of the `[REUSE]` case. Fortunately, the `[REUSE]` case is a `TBSSU` rather than a `TBWU`.

$$\mathbf{init}(t, o_{t'}) = \begin{cases} \boxed{o_{\tilde{t}}} & \text{if } t' \equiv \bullet \text{ [INITIALISE]} \\ o_{t'} & \text{if } \tilde{t} \prec t' \text{ [TBWU]} \\ \boxed{o_{\tilde{t}}} & \text{if } t' \prec \tilde{t} \wedge t' \neq \tilde{t} \text{ [TBSSU]} \\ \boxed{o_{\tilde{t}}} & \text{if } t' \not\prec \tilde{t} \wedge \tilde{t} \not\prec t' \text{ [REUSE]} \\ \emptyset & \text{otherwise [TBSU]} \end{cases}$$

■ **Figure 11** Modifications to the `init` function to account for reuse.

$$\begin{array}{c} \text{[BACK-PROPAGATE-SG]} \\ \hline \ell : p = \&o \quad o_t \text{ newly cloned} \\ \hline o_t \in pt(\ell, p) \\ \\ \text{[BACK-PROPAGATE-FIELD]} \\ \hline \ell : p = \&q \rightarrow f_k \quad \ell' \xrightarrow{q} \ell \quad o \in pt(\ell', q) \\ o' = \mathbf{init}(T(q), o) \quad (o'.f_k)_t \text{ newly cloned} \\ \hline (o'.f_k)_t \in pt(\ell, p) \end{array}$$

■ **Figure 12** Extensions to the analysis to implement back-propagation for stack, global, and field objects.

In the base analysis, back-propagation was only done for heap objects because stack and global objects had one unchanging type (hence they are never cloned). Since C++ allows for reuse of stack and global objects, we need to back-propagate them when they are cloned. [BACK-PROPAGATE-SG] implements this in Figure 12 like the [BACK-PROPAGATE] rule.

Furthermore, we need to account for reuse of field objects. Back-propagation for field objects is less obvious since field objects can be generated at multiple locations depending on the solver’s worklist order. Since fields do not have an allocation site to back-propagate to, field clones are retrieved at any FIELD instruction which had retrieved the original field object, as in the [BACK-PROPAGATE-FIELD] rule. [BACK-PROPAGATE-FIELD] implements this as a second [FIELD] rule operating solely on the clones. Figure 12 also shows this rule.

4.3 Soundness and the heap cloning upper bound

For a C or C++ program conforming to our type model, our analysis is as sound as SPARSE. To soundly analyse programs conforming to our type model, object reuse must be enforced even though doing so incurs a performance and precision penalty. The typical allocation-site-based model bounds the number of objects of a program by the number of allocation sites. Context-sensitive analyses bound the number of context-sensitive heap objects by the number of calling contexts [37]. For real-world scenarios, this is too large, so the context depth is often limited by a small number to make analyses scalable. When the maximum calling context depth is capped at 3 (or more), context-sensitive analysis is usually unscalable for larger programs [37]. The number of heap objects in our analysis is bounded by the number of allocation sites, the number of types on the generated type graph, and the number of fields in the largest structure type. Thus, in the worst case, the number of objects in our analysis would be the product of those three values, which would usually be far fewer than the number of objects created when cloning according to calling contexts.

■ **Table 2** Statistics about the benchmarks. The first column of data represents the lines of code, the second column represents the size of the compiled program’s bitcode, the fourth, fifth, and sixth columns represent the number of different instructions in the bitcode with the number of those instructions which are annotated by `ctir` in parentheses, the seventh column represents the number of canonical types with the number of those which are structs in parentheses, and the final column shows the number of fields in the largest struct in the program.

Bench.	LOC	Size	Instructions (<code>ctir</code> annotated)			# Canon. types (structs)	Largest struct
			Loads	Stores	GEPs		
du	22212	1372 KiB	14742 (2879)	5781 (907)	5384 (4928)	565 (79)	37
date	10002	1132 KiB	12185 (4430)	2860 (912)	7395 (6925)	182 (21)	30
touch	9820	1056 KiB	11416 (4392)	2502 (907)	7304 (6878)	178 (20)	30
ptx	16247	1056 KiB	11787 (2362)	4395 (714)	4546 (4180)	339 (43)	31
csplit	14565	936 KiB	10609 (2020)	3930 (563)	3887 (3628)	347 (49)	31
expr	14070	912 KiB	10336 (2028)	3807 (544)	4000 (3728)	315 (38)	31
tac	13888	876 KiB	10067 (1908)	3678 (491)	3710 (3457)	295 (34)	31
nl	13420	868 KiB	9960 (1924)	3629 (500)	3678 (3469)	293 (34)	31
mv	15962	844 KiB	7713 (1291)	3500 (544)	2437 (2136)	454 (59)	39
ls	14471	804 KiB	7050 (834)	3576 (334)	1655 (1233)	375 (50)	29
ginstall	14968	772 KiB	6843 (944)	3266 (376)	1800 (1521)	416 (53)	39
sort	12000	744 KiB	7743 (945)	3312 (422)	2262 (1802)	391 (58)	42

5 Evaluation

The aim of our evaluation is to compare the performance and precision (through alias testing) of `TYPECLONE` and `SPARSE`. We first describe our implementation of `TYPECLONE`, and all required components, in Section 5.1 and then present the results of our experiments and discuss them in Section 5.2.

5.1 Implementation

Our implementation of the analysis is comprised of two major components: a custom Clang frontend to produce annotated LLVM IR with C/C++ type information and the `TYPECLONE` implementation built upon LLVM and SVF [40]. We use version 9.0 of both Clang and LLVM.

LLVM’s type system is different to that of C/C++. In the LLVM IR produced, Clang does not maintain any type information from C/C++ except through TBAA metadata. Due to the basic nature of TBAA metadata, and that Clang does not annotate all instructions that we are interested in with TBAA metadata (GEP instructions, for example), we implement our own type metadata system called `ctir` which, like EffectiveSan’s customised Clang [14], tags instructions of interest with DWARF debug information which can be read and operated upon by SVF.

During code generation, Clang introduces loads, stores, and other instructions which do not directly map to high-level code. This can, for example, be a byproduct of the nature of partial SSA form, or implementation-defined details like using virtual tables to implement virtual calls. Like TBAA, the instructions and declarations which correspond to C/C++ features of interest are annotated. In the absence of type information, our analysis falls back to standard flow-sensitive pointer analysis methods, not updating upon the type, i.e. not using `init`.

The following are annotated by `ctir`:

- Allocations corresponding to stack and global declarations (the allocated object’s type).
- Load and store instructions which correspond to pointer dereferences and C++ reference accesses (the dereferenced pointer’s element type).
- GEP instructions which correspond to field and array accesses (the base pointer’s element type).
- Virtual calls (the base pointers’s element type).
- Virtual tables (the owning class).

Since DWARF types correspond exactly to C/C++ types, in SVF, we reduce all types to “canonical types” which are types stripped of their signedness, `constexpr`, `typedefs`, and other auxiliary data. The type graph is built from these canonical types and the analysis then only operates on canonical types (converting types obtained from `ctir` annotations as necessary).

A virtual call like $p \rightarrow foo()$ is translated into four LLVM instructions: (1) a `LOAD` instruction, $vtpr = *p$, which retrieves virtual table pointer $vtpr$ by dereferencing pointer p , (2) a `FIELD` instruction, $vfn = \&vtpr \rightarrow k$, which retrieves the entry (i.e., target function) in the virtual table at offset k , (3) a `LOAD` instruction, $fp = *vfn$, which retrieves the address of the target, and finally (4) a `CALL` instruction, $fp(p)$. For calls to external functions where code is unavailable to analyse, a list of commonly used functions is maintained which summarise their side-effects (like `memcpy`, `_Znwm` for C++’s `new`, `mmap`, `strcpy`, and others) following [19, 35].

Within SVF, Andersen’s analysis, optimised with *wave propagation* [34, 29] for better performance, is used to build the value-flow graph of the input program. Our analysis is then implemented on top of the built value-flow graph following the rules described in Section 4. We compare `TYPECLONE` (with and without reuse taken into consideration) with a sparse flow-sensitive and context-insensitive analysis (`SPARSE`) [20] available in SVF [39]. To the best of our knowledge, this is the only publicly available implementation of a whole-program sparse flow-sensitive and context-insensitive C/C++ pointer analysis for LLVM. We also do not know of a publicly available implementation of a whole-program flow- and context-sensitive (`FSCS`) C/C++ pointer analysis for LLVM. According to a study using commercial tools [3], existing `FSCS` algorithms for C “do not scale even for an order of magnitude smaller size programs than those analyzed [with Andersen’s analysis]” in their study. As shown in our evaluation, for annotated pointer accesses, `TYPECLONE` can achieve more precise results than `SPARSE`, thus leaving limited room for benefit in this case by modelling the heap context-sensitively.

5.2 Experiments

We compare the performance and precision of our analysis, with and without reuse considered, with `SPARSE`. We use the 12 largest programs, per LLVM bitcode size, in GNU Coreutils 8.31 (excluding `dir` and `vdirc` since they are almost identical to `ls`). Coreutils was chosen because the included programs use various memory allocation wrappers to perform allocation. Table 2 shows the size (in LOC and of the generated bitcode), number of instructions, number of canonical types and how many of those are structs, and the largest struct by number of fields (after flattening) for each benchmark. All experiments were carried out on a machine running 64-bit Ubuntu 18.04.2 LTS with an Intel Xeon Gold 6132 processor at 2.60GHz and 128GB of memory.

To test the performance, we ran `SPARSE` and `TYPECLONE` (without and with reuse) ten times and averaged the total running time of the analyses (constraint solving upon the VFG, excluding the pre-analyses to build the VFG and other auxiliary data structures like

■ **Table 3** Running times and object counts of SPARSE and TYPECLONE (with and without reuse). The first column of data represents the running time of SPARSE and the second column represents the number of objects in the analysis. The third and fourth columns represent the running time of TYPECLONE (without reuse) and its slowdown from SPARSE, and the fifth column represents the total number of objects in the analysis with the number of clones created in parentheses. The same is repeated for TYPECLONE with reuse in the final 3 columns. The final row shows the geometric mean of slowdown.

Bench.	SPARSE		TYPECLONE			TYPECLONE (reuse)		
	Time	Obj.	Time	Diff.	Obj. (clones)	Time	Diff.	Obj. (clones)
du	15.83s	4295	92.81s	5.86×	5283 (991)	2829.37s	178.73×	7436 (3144)
date	0.34s	1924	1.02s	3.00×	2003 (80)	1.24s	3.65×	2051 (128)
touch	0.33s	1730	0.97s	2.94×	1817 (87)	1.20s	3.64×	1866 (136)
ptx	5.19s	3245	282.76s	54.48×	4242 (997)	378.30s	72.89×	4538 (1292)
csplit	3.45s	2885	4.98s	1.44×	3147 (263)	265.83s	77.05×	3919 (1035)
expr	2.17s	2750	50.40s	23.23×	3358 (609)	80.91s	37.29×	3603 (854)
tac	2.59s	2700	58.84s	22.72×	3383 (684)	78.63s	30.36×	3462 (763)
nl	2.93s	2663	101.69s	34.71×	3342 (680)	118.65s	40.49×	3424 (762)
mv	0.75s	3441	43.90s	58.53×	4403 (961)	67.04s	89.39×	4615 (1173)
ls	0.48s	2975	4.49s	9.35×	3278 (307)	4.58s	9.54×	3302 (331)
ginstall	0.30s	3332	1.75s	5.83×	3712 (378)	2.29s	7.63×	3779 (446)
sort	0.77s	2657	11.93s	15.49×	2994 (339)	12.36s	16.40×	3034 (379)
Average				11.14×			25.19×	

the type graph). The results are presented in Table 3. The “Diff.” columns represent how many times slower an analysis was compared to SPARSE, and the “Obj.” and “Obj. (clones)” columns represent the total number of objects in an analysis, with the number of clones created mentioned separately, where relevant. Generally, we expect running time to increase in TYPECLONE because of the introduction of new objects, which means larger points-to sets and thus extra propagation time, and back-propagation, which means VFG nodes are processed more often. For TYPECLONE without reuse, all slowdown presented is in a general range of acceptability of 1.45×–35× except for when analysing benchmarks **mv** and **ptx**. Both benchmarks created the largest number of clone objects relative to original objects. Overall, the slowdown is usually affordable and the (geometric) mean slowdown is a little over 11× when not considering reuse.

Modelling reuse in TYPECLONE slows down the analysis. Too many opportunities for TBSUs, which would reduce the number of objects created (and prevent some back-propagation) and reduce the size of points-to sets, become TBSSUs with the [REUSE] rule. Stack and global objects also become a source of clones. This is seen in the number of clones created. Benchmarks which were many times slower than TYPECLONE without reuse, like **du** and **csplit**, had many more clones created, and those that remained close in running time had a more modest growth in the number of extra clones created. We see a (geometric) mean slowdown of a little over 25× when considering reuse, more than twice as much compared to the base analysis.

To test the precision, we performed an alias query between all top level pointers of interest within a function (those pointers accessed at an instruction annotated with a C/C++ type with **ctir**) against each other. Two pointers are considered aliases if their points-to sets

■ **Table 4** The number of alias queries performed (between `ctir`-annotated instructions), the number of those alias queries returning a no-alias relation for SPARSE and TYPECLONE (with and without reuse considered), and the improvement to the number of alias queries returning a no-alias relation presented by TYPECLONE (with and without reuse considered) against SPARSE.

Bench.	Queries	SPARSE	TYPECLONE		TYPECLONE (reuse)	
		No-alias results	No-alias results	Improv.	No-alias results	Improv.
du	76291490	55553836	74384866	33.90%	72825202	31.09%
date	151400720	111391680	141377516	26.92%	138450386	24.29%
touch	149194010	109198398	139183292	27.46%	136253192	24.78%
ptx	52845630	43771886	50867888	16.21%	50655276	15.73%
csplit	38719506	30450964	37758260	24.00%	36824196	20.93%
expr	39835032	33654030	38228618	13.59%	38017650	12.97%
tac	34427556	27745100	32782666	18.16%	32654250	17.69%
nl	34863120	27895764	33204888	19.03%	33065830	18.53%
mv	15940056	12655806	14978588	18.35%	14894140	17.69%
ls	6167772	5242862	5869944	11.96%	5817110	10.95%
ginstall	8193906	7755314	7959676	2.64%	7920014	2.12%
sort	10198442	8189034	9583680	17.03%	9499016	16.00%
Average				16.64%		15.36%

intersect, or if one pointer contains a field object generated from an object in the other pointer’s points-to set. A no-alias result is always more desirable than a may-alias result as it paves the way for more optimisations, for example. The number of alias queries, the number of queries returning a no-alias relation (the remainder return a may-alias relations), and the improvement TYPECLONE presents in the number of no-alias relations returned compared to SPARSE are presented in Table 4. We find that for SPARSE, in all benchmarks except `ginstall`, 72%–85% of alias queries return a no-alias result. `ginstall` is an outlier with almost 95% of alias queries returning a no-alias result. For TYPECLONE, 93% to over 97% of alias queries result in a no-alias relation (91% to over 96% when reuse is taken into account). Excluding `ginstall`, TYPECLONE (without considering reuse) sees an increase in almost 12% to almost 34% in the number of no-alias results against SPARSE. The improvement for `ginstall` is much less at 2.64%. The results produced by SPARSE for `ginstall` were already strong and TYPECLONE had little room to improve. Overall, the (geometric) mean improvement sits at over 16% when not taking reuse into consideration. When taking reuse into account, results are still strong albeit weaker than when not taking reuse into account. Excluding `ginstall` again, we see an increase in almost 11% to over 31% in the number of no-alias results against SPARSE. For `ginstall`, the improvement is 2.12%, and the overall (geometric) mean improvement is over 15%.

Overall, TYPECLONE is successful at differentiating points-to sets when objects appear from the same allocation site, as is the case with allocation wrappers, which Coreutils makes use of. We also notice that the imprecision introduced by handling reuse is very slight. Even though handling object reuse eliminates many opportunities for TBSUs, instead creating more clones, the [REUSE] case in `init` is a TBSSU and thus does not cause as much precision loss as it would have if it was a TBWU, even if it may adversely affect performance.

6 Related work

Whole-program flow-sensitive pointer analysis for C and C++ has been studied extensively in the literature. The approaches in [8] and [15] provide the formulations for an iterative data-flow framework [25]. The work presented in [45] considered both flow- and context-sensitivity by representing procedure summaries with partial transfer functions. To eliminate unnecessary propagation of points-to information during the iterative data-flow analysis, sparse analysis propagates points-to facts sparsely across pre-computed def-use chains [20, 33]. Initially, sparsity was achieved through a Sparse Evaluation Graph [9, 21, 22], a refined CFG with irrelevant nodes removed. Further progress was made through various SSA forms like factored SSA [10], HSSA [11] and partial SSA [27]. The def-use chains of top-level pointers, once put in SSA form, can be explicitly and precisely identified, giving rise to a semi-sparse flow-sensitive analysis [19]. Then, by leveraging the idea of staged analyses [17, 20] where a fast, imprecise analysis bootstraps a more precise analysis, flow-sensitive analysis was made fully sparse, with the first stages identifying def-use chains of both top-level and address-taken pointers [20, 39]. Despite these achievements, most flow-sensitive analyses model the heap with one abstract object per allocation site. Most analyses which provide a more precise heap model do so by employing context-sensitivity.

On the other hand, structure-sensitive analysis (`ccllyzer-ss`) [7] improves the allocation-site-based heap model and presents a field-sensitive Andersen’s analysis that lazily infers the types of heap objects through the casting of pointers to those objects to eventually filter out redundant field derivations. When a pointer to a heap object is cast, that object is considered to potentially be of the element type of the pointer it is cast to and so a new object is created with the pointer’s element type, and back-propagated to the allocation site to ensure soundness. Type-based alias analysis (TBAA) [13] uses Modula-3’s type system to (almost) statelessly determine aliasing relations. TBAA is implemented in Clang/LLVM and GCC for C, C++, and Objective-C, and works because of the strict aliasing rules defined by those languages. Inspired by TBAA and `ccllyzer-ss`, this paper proposes a new flow-sensitive type-based heap cloning model to improve the precision of sparse points-to analysis for C and C++ programs which conform to the strict aliasing rules.

7 Conclusion

This paper presents a new flow-sensitive points-to analysis with type-based heap cloning and no context-sensitivity. The novelty of our approach lies in its lazy heap cloning. An untyped abstract heap object created at an allocation site is killed and replaced with a new (clone) object uniquely identified by the type information at its use site for flow-sensitive points-to propagation. This yields more precise points-to relations at different program points without incurring the high costs of context-sensitivity. Our approach also explores a new form of strong updates based on types for flow-sensitive modelling. The resulting analysis improves upon state-of-the-art sparse flow-sensitive analysis answering, on average, over 15% more alias queries with a no-alias result.

References

- 1 ISO/IEC 14882:2017. Programming languages – C++. Standard, International Organization for Standardization, 2017.
- 2 ISO/IEC 9899:2018. Information technology – programming languages – C. Standard, International Organization for Standardization, 2018.

- 3 Mithun Acharya and Brian Robinson. Practical change impact analysis based on static program slicing for industrial software systems. In *2011 33rd International Conference on Software Engineering (ICSE)*, pages 746–755. IEEE, 2011. doi:10.1145/1985793.1985898.
- 4 Lars Ole Andersen. *Program analysis and specialization for the C programming language*. PhD thesis, University of Copenhagen, 1994.
- 5 Dzintars Avots, Michael Dalton, V Benjamin Livshits, and Monica S Lam. Improving software security with a C pointer analysis. In *Proceedings of the 27th International Conference on Software Engineering (ICSE)*, pages 332–341. ACM, 2005. doi:10.1145/1062455.1062520.
- 6 George Balatsouras. *Recovering Structural Information for Better Static Analysis*. PhD thesis, National and Kapodistrian University of Athens, 2017.
- 7 George Balatsouras and Yannis Smaragdakis. Structure-sensitive points-to analysis for C and C++. In *International Static Analysis Symposium (SAS)*, pages 84–104. Springer, 2016. doi:10.1007/978-3-662-53413-7_5.
- 8 Jong-Deok Choi, Michael Burke, and Paul Carini. Efficient flow-sensitive interprocedural computation of pointer-induced aliases and side effects. In *Proceedings of the 20th ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL)*, pages 232–245. ACM, 1993. doi:10.1145/158511.158639.
- 9 Jong-Deok Choi, Ron Cytron, and Jeanne Ferrante. Automatic construction of sparse data flow evaluation graphs. In *Proceedings of the 18th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 55–66. ACM, 1991. doi:10.1145/99583.99594.
- 10 Jong-Deok Choi, Ron Cytron, and Jeanne Ferrante. On the efficient engineering of ambitious program analysis. *IEEE Transactions on Software Engineering*, 20(2):105–114, 1994. doi:10.1109/32.265631.
- 11 Fred Chow, Sun Chan, Shin-Ming Liu, Raymond Lo, and Mark Streich. Effective representation of aliases and indirect memory operations in SSA form. In *International Conference on Compiler Construction (CC)*, pages 253–267. Springer, 1996. doi:10.1007/3-540-61053-7_66.
- 12 Arnab De and Deepak D’Souza. Scalable flow-sensitive pointer analysis for Java with strong updates. In *European Conference on Object-Oriented Programming (ECOOP)*, pages 665–687. Springer, 2012. doi:10.1007/978-3-642-31057-7_29.
- 13 Amer Diwan, Kathryn S McKinley, and J Eliot B Moss. Type-based alias analysis. In *Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation (PLDI)*, page 106–117. ACM, 1998. doi:10.1145/277650.277670.
- 14 Gregory J Duck and Roland HC Yap. EffectiveSan: type and memory error detection using dynamically typed C/C++. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 181–195. ACM, 2018. doi:10.1145/3192366.3192388.
- 15 Maryam Emami, Rakesh Ghiya, and Laurie J Hendren. Context-sensitive interprocedural points-to analysis in the presence of function pointers. *ACM SIGPLAN Notices*, 29(6):242–256, 1994. doi:10.1145/773473.178264.
- 16 Xiaokang Fan, Yulei Sui, Xiangke Liao, and Jingling Xue. Boosting the precision of virtual call integrity protection with partial pointer analysis for C++. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, pages 329–340. ACM, 2017. doi:10.1145/3092703.3092729.
- 17 Stephen J Fink, Eran Yahav, Nurit Dor, G Ramalingam, and Emmanuel Geay. Effective tpestate verification in the presence of aliasing. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 17(2):1–34, 2008. doi:10.1145/1348250.1348255.
- 18 Coreutils - GNU core utilities. URL: <https://www.gnu.org/software/coreutils/>.
- 19 Ben Hardekopf and Calvin Lin. Semi-sparse flow-sensitive pointer analysis. In *Proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, page 226–238. ACM, 2009. doi:10.1145/1594834.1480911.

- 20 Ben Hardekopf and Calvin Lin. Flow-sensitive pointer analysis for millions of lines of code. In *International Symposium on Code Generation and Optimization (CGO)*, pages 289–298. IEEE, 2011. doi:10.1109/CGO.2011.5764696.
- 21 Michael Hind, Michael Burke, Paul Carini, and Jong-Deok Choi. Interprocedural pointer alias analysis. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 21(4):848–894, 1999. doi:10.1145/325478.325519.
- 22 Michael Hind and Anthony Pioli. Assessing the effects of flow-sensitivity on pointer alias analyses. In *International Static Analysis Symposium (SAS)*, pages 57–81. Springer, 1998. doi:10.1007/3-540-49727-7_4.
- 23 Yuseok Jeon, Priyam Biswas, Scott Carr, Byoungyoung Lee, and Mathias Payer. HexType: Efficient detection of type confusion errors for C++. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 2373–2387. ACM, 2017. doi:10.1145/3133956.3134062.
- 24 Sehun Jeong, Minseok Jeon, Sungdeok Cha, and Hakjoo Oh. Data-driven context-sensitivity for points-to analysis. *Proceedings of the ACM on Programming Languages*, 1(OOPSLA):100, 2017. doi:10.1145/3133924.
- 25 John B Kam and Jeffrey D Ullman. Monotone data flow analysis frameworks. *Acta Informatica*, 7(3):305–317, 1977. doi:10.1007/BF00290339.
- 26 George Kastrinis and Yannis Smaragdakis. Hybrid context-sensitivity for points-to analysis. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, page 423–434. ACM, 2013. doi:10.1145/2499370.2462191.
- 27 Chris Lattner and Vikram Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *International Symposium on Code Generation and Optimization (CGO)*, pages 75–86. IEEE, 2004. doi:10.1109/CGO.2004.1281665.
- 28 Anatole Le, Ondřej Lhoták, and Laurie Hendren. Using inter-procedural side-effect information in JIT optimizations. In *International Conference on Compiler Construction (CC)*, pages 287–304. Springer, 2005. doi:10.1007/11406921_22.
- 29 Yuxiang Lei and Yulei Sui. Fast and precise handling of positive weight cycles for field-sensitive pointer analysis. In *International Static Analysis Symposium (SAS)*, pages 27–47. Springer, 2019. doi:10.1007/978-3-030-32304-2_3.
- 30 Ondrej Lhoták and Kwok-Chiang Andrew Chung. Points-to analysis with efficient strong updates. In *Proceedings of the 38th annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 3–16. ACM, 2011. doi:10.1145/1926385.1926389.
- 31 Benjamin Livshits, Manu Sridharan, Yannis Smaragdakis, Ondřej Lhoták, J Nelson Amaral, Bor-Yuh Evan Chang, Samuel Z Guyer, Uday P Khedker, Anders Møller, and Dimitrios Vardoulakis. In defense of soundness: A manifesto. *Communications of the ACM*, 58(2):44–46, 2015. doi:10.1145/2644805.
- 32 V Benjamin Livshits and Monica S Lam. Tracking pointers with path and context sensitivity for bug detection in C programs. In *Proceedings of the 9th European Software Engineering Conference held jointly with 11th ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE)*, pages 317–326. ACM, 2003. doi:10.1145/940071.940114.
- 33 Hakjoo Oh, Kihong Heo, Wonchan Lee, Woosuk Lee, and Kwangkeun Yi. Design and implementation of sparse global analyses for C-like languages. In *Proceedings of the 33rd ACM SIGPLAN conference on Programming Language Design and Implementation (PLDI)*, pages 229–238. ACM, 2012. doi:10.1145/2345156.2254092.
- 34 Fernando Magno Quintao Pereira and Daniel Berlin. Wave propagation and deep propagation for pointer analysis. In *2009 International Symposium on Code Generation and Optimization (CGO)*, pages 126–135. IEEE, 2009. doi:10.1109/CGO.2009.9.
- 35 Rajiv Ravindran Rick Hank, Loreena Lee. Implementing next generation points-to in Open64. In *Open64 Developers Forum*, 2010. URL: <http://www.affinic.com/documents/open64workshop/2010/>.

- 36 Qingkai Shi, Xiao Xiao, Rongxin Wu, Jinguo Zhou, Gang Fan, and Charles Zhang. Pinpoint: Fast and precise sparse value flow analysis for million lines of code. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 693–706. ACM, 2018. doi:10.1145/3192366.3192418.
- 37 Yannis Smaragdakis, Martin Bravenboer, and Ondrej Lhoták. Pick your contexts well: understanding object-sensitivity. In *Proceedings of the 38th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL)*, pages 17–30. ACM, 2011. doi:10.1145/1925844.1926390.
- 38 Johannes Späth, Lisa Nguyen Quang Do, Karim Ali, and Eric Bodden. Boomerang: Demand-driven flow- and context-sensitive pointer analysis for Java. In *30th European Conference on Object-Oriented Programming (ECOOP)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2016. doi:10.4230/LIPIcs.ECOOP.2016.22.
- 39 Yulei Sui and Jingling Xue. On-demand strong update analysis via value-flow refinement. In *ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, pages 460–473. ACM, 2016. doi:10.1145/2950290.2950296.
- 40 Yulei Sui and Jingling Xue. SVF: interprocedural static value-flow analysis in LLVM. In *Proceedings of the 25th International Conference on Compiler Construction (CC)*, pages 265–266. ACM, 2016. doi:10.1145/2892208.2892235.
- 41 Yulei Sui, Ding Ye, and Jingling Xue. Detecting memory leaks statically with full-sparse value-flow analysis. *IEEE Transactions on Software Engineering*, 40(2):107–122, 2014. doi:10.1109/TSE.2014.2302311.
- 42 Frank Tip. *A survey of program slicing techniques*. Centrum voor Wiskunde en Informatica Amsterdam, 1994.
- 43 Mark Weiser. Program slicing. In *Proceedings of the 5th International Conference on Software Engineering (ICSE)*, page 439–449. IEEE, 1981. doi:10.1109/TSE.1984.5010248.
- 44 John Whaley and Monica S Lam. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In *Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation (PLDI)*, pages 131–144. ACM, 2004. doi:10.1145/996841.996859.
- 45 Robert P Wilson and Monica S Lam. Efficient context-sensitive pointer analysis for C programs. *ACM Sigplan Notices*, 30(6):1–12, 1995. doi:10.1145/223428.207111.

Scala with Explicit Nulls

Abel Nieto 

University of Waterloo, Canada
anietoro@uwaterloo.ca

Yaoyu Zhao

University of Waterloo, Canada
y437zhao@edu.uwaterloo.ca

Ondřej Lhoták 

University of Waterloo, Canada
olhotak@uwaterloo.ca

Angela Chang

University of Waterloo, Canada
yue.chang@edu.uwaterloo.ca

Justin Pu

University of Waterloo, Canada
justin.pu@edu.uwaterloo.ca

Abstract

The Scala programming language makes *all* reference types *implicitly nullable*. This is a problem, because `null` references do not support most operations that do make sense on regular objects, leading to runtime errors. In this paper, we present a modification to the Scala type system that makes nullability *explicit* in the types. Specifically, we make reference types *non-nullable* by default, while still allowing for nullable types via *union types*. We have implemented this design for explicit nulls as a fork of the Dotty (Scala 3) compiler. We evaluate our scheme by migrating a number of Scala libraries to use explicit nulls. Finally, we give a *denotational semantics* of *type nullification*, the interoperability layer between Java and Scala with explicit nulls. We show a soundness theorem stating that, for variants of System F_{ω} that model Java and Scala, nullification preserves values of types.

2012 ACM Subject Classification Software and its engineering → General programming languages; Theory of computation → Denotational semantics; Theory of computation → Type theory; Software and its engineering → Interoperability

Keywords and phrases Scala, Java, nullability, language interoperability, type systems

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2020.25

Supplementary Material ECOOP 2020 Artifact Evaluation approved artifact available at <https://doi.org/10.4230/DARTS.6.2.14>.

Funding This research was supported by the Natural Sciences and Engineering Research Council of Canada and by the Waterloo-Huawei Joint Innovation Lab.

Acknowledgements We would like to thank Sébastien Doeraene, Fengyun Liu, Guillaume Martres, and Martin Odersky for their feedback on our explicit nulls design.

1 Introduction

Scala inherited elements of good design from Java, but it also inherited at least one misfeature: the `null` reference. In Scala, like in many other object-oriented programming languages, the `null` reference can be typed with *any reference type*. This leads to runtime errors, because



© Abel Nieto, Yaoyu Zhao, Ondřej Lhoták, Angela Chang, and Justin Pu; licensed under Creative Commons License CC-BY

34th European Conference on Object-Oriented Programming (ECOOP 2020).

Editors: Robert Hirschfeld and Tobias Pape; Article No. 25; pp. 25:1–25:26

Leibniz International Proceedings in Informatics



LIPICs Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany



25:2 Scala with Explicit Nulls

`null` does not (and cannot) support almost *any* operations. For example, the program below tries to read the `length` field of a string, only to find out that the underlying reference is `null`. The program then terminates with the infamous `NullPointerException`¹.

```
val s: String = null           // ok: String is a nullable type
println(s" s has length " + s.length) // throws a NullPointerException
```

Errors of this kind are very common, and can sometimes lead to security vulnerabilities. Indeed, “Null Pointer Dereference” appears in position 14 of the *2019 CWE Top 25 Most Dangerous Software Errors*, a list of vulnerability classes maintained by the MITRE Corporation [21]. As of November 2019, a search for “null pointer dereference” in MITRE’s vulnerability database² returned 1429 entries.

The root of the problem lies in the way that Scala structures its type hierarchy. The `null` reference has type `Null`, and `Null` is considered to be a subtype of any reference type. In the example above, `Null` is a subtype of `String`, and so the initializer `val s: String = null` is allowed. We could say that in Scala, (reference) types are *implicitly nullable*. The alternative is to have a language where nullability has to be *explicitly* indicated. For example, we can re-imagine the previous example in a system with explicit nulls (the notation `String|Null` stands for the *union type* “String or Null”):

```
val s: String = null           // error: Null is not a subtype of String
val s: String|Null = null      // ok: s is explicitly marked as nullable
println("s has length " + s.length) // error: String|Null does not have a 'length' field
if (s != null) println("s has length " + s.length) // ok: we checked that s is not null
```

In a world with explicit nulls, the type system can keep track of which variables are potentially `null`, turning runtime errors into compile-time errors.

Our contributions, implemented on top of the Dotty (Scala 3) compiler and currently under consideration for inclusion in Scala 3, are as follows:

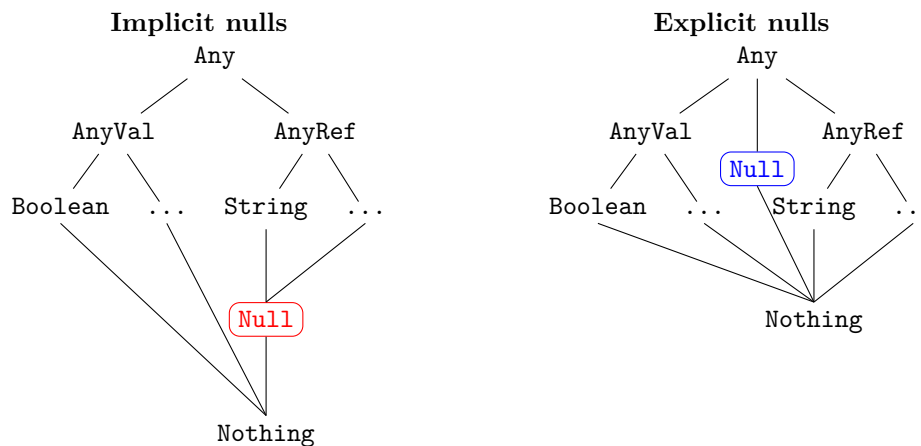
- We retrofitted Scala’s type system with a mechanism for tracking nullability, using union types. To improve usability of nullable values in Scala code, we also added a simple form of flow typing to Scala.
- So that Scala programs can interoperate with Java code, where `nulls` remain implicit, we present a *type nullification* function that turns Java types into equivalent Scala types.
- We evaluate the design by migrating multiple Scala libraries to explicit nulls. The main findings are that most of the effort in migrating Scala code to explicit nulls comes from Java interoperability, and that the effort is significant for some libraries.
- Finally, we formalize type nullification using variants of System F_ω that have been augmented to model implicit and explicit nulls. Using denotational semantics, we prove a *soundness* theorem for nullification, saying that nullification preserves values of types.

2 A New Type Hierarchy

To understand the special status of the `Null` type, we can inspect the current Scala type hierarchy, shown in Figure 1. Roughly, Scala types can be divided into *value* types (subtypes of `AnyVal`) and *reference* types (subtypes of `AnyRef`). The type `Any` then stands at the top of the hierarchy, and is a supertype of both `AnyVal` and `AnyRef` (in fact, a supertype of

¹ <https://docs.oracle.com/javase/8/docs/api/java/lang/NullPointerException.html>

² <https://cve.mitre.org/cgi-bin/cvekey.cgi?keyword=NULL+Pointer+Dereference>



■ **Figure 1** Alternative Scala type hierarchies with implicit and (our design) explicit nulls.

every other type). Conversely, `Nothing` is a subtype of all types. Finally, `Null` occupies an intermediate position: it is a subtype of all *reference* types, but not of the value types. This justifies the following typing judgments:

```
val s: String = null // ok: String is a reference type
val i: Int = null // error: Int is a value type
```

This is what makes `nulls` in Scala *implicit*. In order to make `nulls` *explicit*, we need to dislodge the `Null` type from its special position, so that it is no longer a subtype of all reference types. We achieve this by making `Null` a direct subtype of `Any`. This new type hierarchy, which underlies our design, is also shown in Figure 1. With the new type hierarchy we get new typing judgments:

```
val s: String = null // error: reference types like String are no longer nullable
val i: Int = null // error: Int is a value type
val sn: String|Null = null // ok: Null <: String|Null
```

`String|Null` is a *union type*. In general, the union type `A|B` (read “A or B”) contains *all* values of both `A` and `B`, as indicated by the subtyping judgements `A <: A|B` and `B <: A|B`. Union types are a new feature present in Dotty but not in Scala 2 and, as the example shows, they allow us to encode nullability.

The explicit nulls hierarchy is still unsound in the presence of uninitialized values:

```
1 class Person {
2   val name: String = getName()
3   def getName(): String = "Person" + name.length // 'name' is null here
4 }
5 val p = new Person() // throws a NullPointerException
```

Because, after allocation, the fields of Scala classes are initialized to their “default” values, and the default value for reference types is `null`, when we try to access `name.length` in line 3, `name` is `null`. This produces a `NullPointerException`. While ensuring sound initialization is an interesting challenge, it is not one we tackle in this paper. Developing a sound initialization scheme for Scala, while balancing soundness with expressivity, remains future work. We review some of the existing approaches in Section 7.

2.1 Fixing a Soundness Hole

Even though explicit nulls do not make the Scala type system sound (e.g. there remain null-related soundness holes related to incomplete initialization of class fields), they *do* remove the specific source of unsoundness identified by Amin and Tate [3]. This class of bugs, reported in 2016 and still present in Scala and Dotty, happens due to a combination of implicit nullability and type members with arbitrary lower and upper bounds. For example, the example presented by Amin and Tate [3] crucially relies on being able to construct a term t that has *both* type e.g. `LowerBound[Int]` and `UpperBound[String]`, two type applications unrelated by subtyping. Because of implicit nullability, `null` has both types, which makes the unsoundness possible. With our explicit nulls design, the typing above is no longer possible, so the runtime error becomes a compile-time error.

3 Java Interoperability

One of Scala’s strengths is its ability to seamlessly use Java libraries. Because both languages are compiled down to Java Virtual Machine (JVM) *bytecode* [18], Java libraries appear to Scala code as any other Scala library would. The interaction can also happen in the opposite direction: Java code can use Scala libraries.

Because reference types remain implicitly nullable in Java, we need a way to “interpret” Java types as Scala types, where nullability is explicit. For example, if a Java method returns a `String`, then the Java type system will allow `null` as a return value. If we use said method from Scala, we need to interpret the method’s type as `String|Null`.

In the opposite direction, when Java code uses Scala libraries, the problem is simpler because Java types are less precise than Scala types. In particular, both the Scala types `String` and `String|Null` can be interpreted as the Java type `String`, which includes the null value.

3.1 Type Nullification

Type nullification is the process of translating Java types to their Scala equivalents, in the presence of explicit nulls. By equivalent, we mean that if type nullification sends type A to type B , the values of A and B must be the same. Below are two examples of the behaviour we want from nullification:

- The values of the `StringJava` type³ are all finite-length strings (e.g. `"hello world"` and `"`), plus the value `null`. By contrast, the values of `StringScala` are *just* all finite-length strings (but not `null`). This means that nullification must map `StringJava` to `StringScala|Null`.
- Similarly, we can think of a Java method with signature `StringJava getName(StringJava s)` as representing a function from `StringJava` to `StringJava` (i.e. `getName: StringJava → StringJava`). Suppose that $f \in \text{String}_{\text{Java}} \rightarrow \text{String}_{\text{Java}}$. Notice that f can take `null` as an argument, and return `null` as a result. This means that nullification should return `StringScala|Null → StringScala|Null` in this case.

Here is why “preserves values of a type” is a useful correctness criterion for nullification. Suppose that nullification instead *underapproximated* a type’s values. For example, we could turn `StringJava` into `StringScala`. We might then call e.g. the `length` method on the

³ We write T_{Java} and T_{Scala} for Java’s and Scala’s view of the same type T , respectively.

$F_{\text{null}}(R) = R \text{Null}$	if R is a reference type	(FN-Ref)
$F_{\text{null}}(R) = R$	if R is a value type	(FN-Val)
$F_{\text{null}}(T) = T \text{Null}$	if T is a type parameter	(FN-Par)
$F_{\text{null}}(C\langle R \rangle) = C\langle A_{\text{null}}(R) \rangle \text{Null}$	if C is Java-defined	(FN-JG)
$F_{\text{null}}(C\langle R \rangle) = C\langle F_{\text{null}}(R) \rangle \text{Null}$	if C is Scala-defined	(FN-SG)
$F_{\text{null}}(A\&B) = (A_{\text{null}}(A)\&A_{\text{null}}(B)) \text{Null}$		(FN-And)
$A_{\text{null}}(R) = R$	if R is a reference type	(AN-Ref)
$A_{\text{null}}(T) = T$	if T is a type parameter	(AN-Par)
$A_{\text{null}}(C\langle R \rangle) = C\langle A_{\text{null}}(R) \rangle$	if C is Java-defined	(AN-JG)
$A_{\text{null}}(C\langle R \rangle) = C\langle F_{\text{null}}(R) \rangle$	if C is Scala-defined	(AN-SG)
$A_{\text{null}}(R) = F_{\text{null}}(R)$	otherwise	(AN-FN)

F_{null} is applied to the types of fields, and argument and return types of methods of *every* Java-defined class. We try the rules in top-to-bottom order, until one matches.

■ **Figure 2** Type nullification functions.

`StringScala`, only to find out that the underlying reference was `null`. Another way of saying this is that underapproximations are *unsound for reads*. Similarly, consider what would happen were nullification to *overapproximate* types. For example, we could map `StringJava` to Scala’s `Any`. This is sound for reads, because we cannot call `length` on an `Any`. However, if the Java type were to appear *contravariantly*, e.g. as a method argument, then the Scala code could pass an `Any` (a value of any type), where the Java code expects a `StringJava`, leading to runtime errors. That is, overapproximations are *unsound for writes*. This leads us back to our desired goal of preserving values of types. For now, we only argue informally that nullification preserves values of types. Section 6 formalizes this idea using denotational semantics and proves soundness of the rules on a core calculus.

Nullification can be described with a pair of mutually-recursive functions ($F_{\text{null}}, A_{\text{null}}$) that map Java types to Scala types. The functions are defined in Figure 2 and described below. But first, a word about how nullification is applied. The Dotty compiler can load Java classes in two ways: from source or from bytecode. In either case, when a Java class is loaded, we apply F_{null} to the types of fields and the argument and result type of methods. The resulting class with modified fields and methods is then made accessible to the Scala code. Below is some intuition and example for the different nullification rules.

Case (FN-Ref and FN-Val) These two rules are easy: we nullify reference types but not value types, because *only* reference types are nullable in Java. Here is an example Java class and its translation (given in Java syntax enhanced with union types and a `Null` type):

<pre>// Java class class C { String s; int x; }</pre>	<pre>// After nullification class C { String Null s; int x; }</pre>
---	---

Case (FN-Par) Since type parameters are always nullable in Java, we need to nullify them as well.

25:6 Scala with Explicit Nulls

<pre>// Java class class C<T> { T foo() {...} }</pre>	<pre>// After nullification class C<T> { T Null foo() {...} }</pre>
---	---

For example, if we have `c: C<Boolean>`, then `c.foo()` now returns a `Boolean|Null`, as opposed to just `Boolean` like it used to.

Case (FN-JG) This rule handles generics `C<T>`, where `C` is a class defined in Java (*Java-defined*). The rule is designed to reduce the number of redundant nullable types we need to add. Let us look at an example:

<pre>// Java class Box<T> { T get(); } class BoxFactory<T> { Box<T> makeBox(); }</pre>	<pre>// After nullification class Box<T> { T Null get(); } class BoxFactory<T> { Box<T> Null makeBox(); }</pre>
--	---

Suppose we have a `BoxFactory<String>`. Notice that calling `makeBox` on it returns a `Box<String>|Null`, not a `Box<String|Null>|Null`, because of **FN-JG**. This seems at first glance unsound, because the box itself could contain `null`. However, it is sound because calling `get` on a `Box<String>` returns a `String|Null`.

Generalizing from the example, we can see that it is enough to nullify the type application `C<T>` as `C<T>|Null`. That is, it is enough to mark the type as nullable only at the top level, since uses of `T` in the body of `C` will be nullified as well, *if C is Java-defined*. Notice that the correctness argument relies on our ability to patch all Java-defined classes that transitively appear in the argument or return type of a field or method accessible from the Scala code being compiled. All such classes must be visible to the Scala compiler in any case, and will thus be nullified, so this requirement is satisfied by the implementation.

In fact, the rule is a bit more complicated than we have explained so far. The full rule is $F_{\text{null}}(C<R>) = C<A_{\text{null}}(R)>|Null$. Notice that in fact we *do* transform the type argument, but do so using A_{null} instead of F_{null} . A_{null} is a version of F_{null} that does not add `|Null` at the top level. A_{null} is needed for cases where we have nested type applications, and it is explained in more detail below. Here is a sample application of F_{null} to a nested type application, assuming that `C`, `D`, and `String` are all Java-defined:

$$\begin{aligned}
 F_{\text{null}}(C<D<String>>) &= C<A_{\text{null}}(D<String>>)|Null \\
 &= C<D<A_{\text{null}}(String)>>|Null \\
 &= C<D<String>>|Null
 \end{aligned}$$

Notice how we only add `|Null` at the outermost level. This minimizes the number of changes required to migrate existing Scala code with Java dependencies.

Case (FN-SG) This rule handles the mirror case, where the Java code refers to a generic `C<T>` in which `C` is a class defined in Scala (*Scala-defined*). For example, assuming that `Box` is Scala-defined, we get:

<pre>// Java code that refers to Scala class Box class BoxFactory<T> { Box<T> makeBox(); }</pre>	<pre>// After nullification class BoxFactory<T> { Box<T Null> Null makeBox(); }</pre>
--	---

Notice that unlike the previous rule, **FN-SG** adds `|Null` to the type argument, and not just to the top level. This is needed because nullification is only applied to Java classes, and not to Scala classes. We then need a way to indicate that, in the example, the returned `Box` may contain `null`.

Case (FN-And) This rule just recurses structurally on the components of the type. Even though Java does not have intersection types, we sometimes encounter them during nullification, because the Scala compiler desugars some Java types using intersections. For example, the Java type `Array[T]`, where `T` has no supertype, is represented in Scala as `Array[T & Object]`.

As previously mentioned, A_{null} is a helper function that behaves mostly like F_{null} , but never nullifies types at the top level. A_{null} is useful because we want to avoid adding superfluous `|Null` unions whenever possible.

4 Flow Typing

To improve usability of nullable types, we added a simple form of flow-sensitive type inference to Scala [15]. The general idea is that sometimes, by looking at the control flow, we can infer that a value previously thought to be nullable (due to its type) is no longer so.

4.1 Supported Cases

Below we list the cases supported by flow typing. In the examples, the notation `???` stands for an unspecified expression of the appropriate type⁴:

- *Branches of an if-expression.* If an if-expression has a condition `s != null`, where `s` satisfies some restrictions (see below), then in the `then` branch we can assume that `s` is non-nullable.

```
val s: String|Null = ???
if (s != null) {
  val l = s.length // ok: s has type String in the 'then' branch
}
val l = s.length // error: s has type String|Null
```

We can reason similarly about the `else` branch if the test is `p == null`.

- *Logical operators.* We also support the logical operators `&&`, `||`, and `!` in conditions: e.g. given a condition `if (s != null && s2 != null)`, we infer that *both* `s` and `s2` are non-null in the `then` branch.
- *Propagation within conditions.* We support type specialization within a condition, taking into account that `&&` and `||` are short-circuiting: e.g. in the condition `s != null && s.length > 0`, the test `s.length` is type correct because the right-hand side of the condition will only be evaluated if `s` is non-null.
- *Nested conditions.* Our inference works in the presence of arbitrarily-nested conditions: given the condition `!(a = null || b = null) && (c != null)`, we infer that all of `a`, `b`, and `c` are non-null in the `then` branch.
- *Early exit from blocks.* If a statement conditionally performs an early exit from a block based on whether a value is `null`, we can soundly assume that the value is non-null from that point on. This is the case for both `return` statements and exceptions:

```
if (s == null) return 0
return s.length // ok: s inferred to have type String from this point on
```

In general, if we have a block `s1, ..., si, si+1, ..., sn`, where the `si` are statements, and `si` is of the form `if (cond) exp`, where `exp` has type `Nothing`, then depending on `cond`, we might be able to infer additional nullability facts for statements `si+1, ..., sn`. Here,

⁴ `???` is actually valid Scala code, and is simply a method with return type `Nothing`.

the condition `cond` can contain nested conditions such as those discussed in the previous point. The reason is that type `Nothing` has no values, so an expression of type `Nothing` cannot terminate normally (it either throws or loops). It is then safe to assume that statement s_{i+1} executes only if `cond` is `false`.

There is one extra complication here, which is that Scala allows forward references to method definitions, which combined with nested methods can lead to non-intuitive control flow. In our implementation, we have logic for detecting forward references and disabling flow typing in such cases to preserve soundness. In the presence of forward references, we discard the more precise type inferred for a specific program point by flow typing and fall back to the flow-insensitive declared type that is conservatively sound at all program points.

4.1.1 Stable Paths

Scala has four kinds of definitions: `vals`, `lazy vals`, `vars`, and `defs`. `vals` are eagerly evaluated and immutable. `lazy vals` are like `vals`, but lazily evaluated and then memoized. `vars` are eagerly evaluated and mutable. Finally, `defs` are lazily evaluated, but not memoized, so they are used to define methods.

We use flow typing on `vals` and `lazy vals`, but not on `vars` or `defs`. Using naive flow typing on a `var` would be unsound, because the underlying value might change between the moment it is tested (where it might be non-null) and the later use of the `var` (where it might be again `null`). Similarly, flow typing on `defs` would be problematic, because a `def` is not guaranteed to return the same value after every invocation.

In general, given a path $p = v.s_1.s_2.\dots.s_n$, where v is a local or global symbol, and the s_i are selectors, it is safe to do flow inference on p *only if* p is *stable*. That is, all of v, s_1, \dots, s_n need to be `vals` or `lazy vals`. If p is stable, then we know that p is immutable and so the results of a check against `null` are persistent and can be trusted.

4.2 Inferring Flow Facts

The goal of flow typing is to discover *nullability facts* about stable paths that are in scope. A *fact* is an assertion that a specific path is non-null at a given program point.

At the core of flow typing, we have a function $\mathcal{N} : \text{Exp} \times \text{Bool} \rightarrow \mathbb{P}(\text{Path})$. \mathcal{N} takes a Scala expression e (where e evaluates to a boolean) and a boolean b , and returns a set of paths known to be non-nullable if e evaluates to b . That is, $\mathcal{N}(e, \text{true})$ returns the set of paths that are non-null if e evaluates to `true`, and $\mathcal{N}(e, \text{false})$ returns the set of paths known to be non-null if e evaluates to `false`. \mathcal{N} is defined in Figure 3.

We can use \mathcal{N} to support the flow typing scenarios we previously outlined:

- Given an if expression `if (cond) e1 else e2`, we compute $F_{\text{then}} = \mathcal{N}(\text{cond}, \text{true})$ and $F_{\text{else}} = \mathcal{N}(\text{cond}, \text{false})$. The former gives us a set of paths that are known to be non-null if `cond` is true. This means that we can use F_{then} when typing e_1 . Similarly, we can use F_{else} when typing e_2 .
- To reason about nullability *within* a condition `e1 && e2`, notice that e_2 is evaluated *only* if e_1 is `true`. This means that we can use the facts in $\mathcal{N}(e_1, \text{true})$ when typing e_2 . Similarly, in a condition `e1 || e2`, we only evaluate e_2 if e_1 is *false*. Therefore, we can use $\mathcal{N}(e_1, \text{false})$ when typing e_2 .
- Given a block with statements `if (cond) e; s`, where e has type `Nothing`, or a block of the form `if (cond) return; s`, we know that s will only execute if `cond` is false. Therefore, we can use $\mathcal{N}(\text{cond}, \text{false})$ when typing s .

$$\begin{aligned}
\mathcal{N}(p == \text{null}, \text{true}) &= \{\} \\
\mathcal{N}(p == \text{null}, \text{false}) &= \{p\} \text{ if } p \text{ is stable} \\
\mathcal{N}(p != \text{null}, \text{true}) &= \{p\} \text{ if } p \text{ is stable} \\
\mathcal{N}(p != \text{null}, \text{false}) &= \{\} \\
\mathcal{N}(A \ \&\& \ B, \text{true}) &= \mathcal{N}(A, \text{true}) \cup \mathcal{N}(B, \text{true}) \\
\mathcal{N}(A \ \&\& \ B, \text{false}) &= \mathcal{N}(A, \text{false}) \cap \mathcal{N}(B, \text{false}) \\
\mathcal{N}(A \ || \ B, \text{true}) &= \mathcal{N}(A, \text{true}) \cap \mathcal{N}(B, \text{true}) \\
\mathcal{N}(A \ || \ B, \text{false}) &= \mathcal{N}(A, \text{false}) \cup \mathcal{N}(B, \text{false}) \\
\mathcal{N}(!A, \text{true}) &= \mathcal{N}(A, \text{false}) \\
\mathcal{N}(!A, \text{false}) &= \mathcal{N}(A, \text{true}) \\
\mathcal{N}(\{s1; \dots; sn; \text{cond}\}, b) &= \mathcal{N}(\text{cond}, b) \\
\mathcal{N}(e, b) &= \{\} \text{ otherwise}
\end{aligned}$$

■ **Figure 3** Flow facts inference. Correctness follows from De Morgan’s laws.

4.3 Asserting Non-Nullability

For cases where flow typing is not powerful enough to infer non-nullability, we added a `.nn` (“assert non-nullable”) method to cast away nullability from any term.

```

var s: String | Null = ???
val l = s.nn.length // ok: .nn method casts away nullability

```

In general, if `e` is an expression with type `T | Null`, then `e.nn` has type `T`. The `nn` method is defined as an *extension method*. This is a kind of implicit definition that makes `nn` available for any receiver of type `T | Null`. `nn` does a *checked cast*, so `e.nn` fails with an exception if the receiver `e` evaluates to `null`.

5 Evaluation

In this section, we empirically evaluate the expressiveness of the explicit nulls system and the effort required to migrate existing Scala programs to it. We test the popular belief that Scala programs tend not to use `null` references much themselves except for interaction with Java code. The explicit nulls system requires a program to explicitly specify what is to be done if a `null` reference arises at each program location where it is not ruled out statically; we quantify how many such locations there are in typical Scala programs.

We perform our evaluation on the programs in the Dotty community build,⁵ a suite of Scala programs that have been ported from Scala 2 to compile with the Dotty compiler (without explicit nulls), and are regularly tested as part of the Dotty regression tests. The community build programs are summarized in Table 1.

We divide our evaluation into three parts. First, in Section 5.1, we evaluate `null` references possibly coming from interaction with Java code. Second, in Section 5.2, we evaluate the effectiveness of flow-sensitive typing in ruling out the possibility of `null` references. Third, in Section 5.3, we examine other causes of null-related compilation errors that are not related to interaction with Java and are not ruled out by flow typing.

⁵ <https://github.com/lampepfl/dotty/tree/master/community-build/test/scala/dotty/communitybuild>

■ **Table 1** Community build libraries.

Name	Description	Size (LOC)	Files
scala-pb	Scala protocols buffer compiler	37,029	275
squants	DSL for quantities	14,367	222
fastparse	Parser combinators	13,701	80
effpi	Verified message passing	5,760	60
betterfiles	IO library	3,321	29
algebra	Algebraic type classes	3,032	75
scopt	Command-line options parsing	3,445	28
shapeless	Type-level generic programming	2,328	18
scalap	Class file decoder	2,210	22
semanticdb	Data model for semantic information	2,154	49
intent	Test framework	1,866	48
minitest	Test framework	1,171	32
xml-interpolator	XML string interpolator	993	20
stdLib213	Scala standard library	31,723	588
scala-xml	XML support	6,989	115
scalactic	Utility library	3,952	53
Total		134,041	1,714

5.1 Evaluation of Java interaction

We evaluate the interaction with Java code by counting the number of compilation errors in several variants of the explicit nulls system. The error counts per thousand lines of code for each program and each variant are shown in Table 2.

The **Baseline** column shows the error counts for the explicit nulls system as described in this paper so far. There is significant variance between the different programs, from two or fewer errors per thousand lines of code in more abstract, Scala-like programs, to tens of errors per thousand lines of code in more low-level programs, particularly those that interact significantly with Java. We conjecture that interaction with Java is the main cause of the errors, and evaluate several variations of the system to test this conjecture.

Our first attempt to reduce the number of errors is with nullness annotations in Java code. The **Annotations** column shows the error counts when the Scala programs are compiled with a variant of the Java standard library with annotations specifying that the return values of certain methods cannot be `null`. The annotations are taken from the Checker Framework Project [23], which publishes an annotated version of the Java standard library, with nullness annotations on 4414 methods and 1712 fields in 847 classes. There are many different standards for annotating Java code with nullability; our implementation supports reading 12 such annotation formats and additional formats can be added easily. On some of the programs with high error counts, the annotations reduce the error count significantly, by up to half on `scalap`, but on others, they make little difference, such as on `ScalaPB`. One reason for this is that some programs interact with Java code other than the standard library, and the other Java libraries are not annotated. Another reason is that although the Checker Framework provides thousands of annotations, it still leaves a large part of the standard library unannotated, and the Scala programs interact with these unannotated methods. Annotating the entire standard library would be a huge effort, and even then, more annotations would be needed for any other Java libraries that a Scala program interacts with.

■ **Table 2** Error frequency by configuration in errors per thousand LOC. The mean is weighted by the number of LOC in each program. The **Baseline** column reflects the configuration described in this paper so far. The **Annotations** column adds annotations to the Java standard library to specify methods that do not return `null`. The **JavaNull** column reflects a configuration in which method selections are (unsoundly) allowed on possibly `null` references returned by Java methods. The **Non-null Ret.** column reflects a configuration in which all calls of Java methods are (unsoundly) assumed to never return a `null` reference. The **Ann. No Flow** column reflects a configuration like the **Annotations** column, except with the flow typing discussed in Section 4 disabled.

	Baseline	Annotations	JavaNull	Non-null Ret.	Ann. No Flow
scalactic	72.37	57.19	57.19	3.04	57.19
betterfiles	43.36	38.54	37.04	7.23	38.54
stdLib213	37.26	34.01	33.54	17.24	34.36
ScalaPB	24.98	24.76	24.76	1.38	24.76
minitest	18.79	13.66	12.81	6.83	13.66
scalap	15.84	7.69	7.24	1.81	7.69
scala-xml	13.59	11.45	11.30	9.30	11.88
semanticdb	12.07	7.43	6.04	1.39	7.43
intent	8.57	6.97	6.97	0.54	6.97
scopt	5.52	4.93	4.64	2.32	4.93
xml-interpolator	2.01	2.01	2.01	2.01	2.01
shapeless	1.72	0.00	0.00	0.00	0.00
fastparse	1.61	1.53	1.53	1.46	1.53
effpi	1.39	1.39	1.04	0.00	1.56
algebra	0.33	0.33	0.33	0.00	0.33
squants	0.00	0.00	0.00	0.00	0.00
Mean	20.79	18.96	18.74	5.56	19.07

Another conjecture is that it is common to chain calls to Java methods. For example, if `s` is of type `String`, we may call `s.trim.toUpperCase`, where `trim` is a Java method on strings that returns another string, on which we wish to call the Java method `toUpperCase`. Such a pattern is rejected by the explicit nulls system if `trim` can return a null reference, since a null reference does not have a `toUpperCase` method, but if this pattern is common, it may be pragmatic to allow it, even if it is potentially unsound. We evaluate a variant of the explicit nulls system that adds a special `JavaNull` annotation to mark `Null` types returned from Java methods. The Dotty type system treats these annotated `Null` types the same as any other `Null` types, with the exception that a method in a class `C` can be called on a receiver of type `C | Null` if the `Null` has the special annotation. This variant permits the sequence of calls `s.trim.toUpperCase`, since the nullable return type of `trim` has the special `JavaNull` annotation. Note that this pragmatic design decision sacrifices soundness. The error counts for this variant of the explicit nulls system, together with the standard library annotations from the Checker Framework, are shown in the **JavaNull** column. Although the `JavaNull` annotation does reduce error counts for some programs, the reduction is small. This suggests that there are important things other than method selections that Scala programs do with the values returned from Java methods, and thus the `JavaNull` annotation to enable method selections is not sufficient to significantly reduce error counts.

Finally, we measure an upper bound on the reduction in error count that can be achieved by annotating Java methods that return non-null values. We evaluate a configuration of the explicit nulls system that assumes that *every* call to a Java method returns a non-null

value. This is equivalent to annotating every possible Java method with a non-null return type annotation. It is also equivalent to an extreme case of the special `JavaNull` annotation, which exceptionally allows method selection on nullable values returned from Java methods: if we were to allow `JavaNull` types in all places that currently require non-null types, rather than only in method selections, this would be equivalent to assuming that return values of Java methods cannot be `null`.

The resulting error counts are shown in the **Non-null Ret.** column. The impact of this configuration is very large: it causes a major reduction in error counts in all of the programs that still have large numbers of compilation errors. The `scalactic` library, which had over 72 errors per thousand lines of code in the baseline configuration, has only just over 3 errors per thousand lines of code. The mean error count goes from about 21 in the Baseline configuration and about 19 in the Annotations and `JavaNull` configurations down to 6 in the Non-null Ret. configuration. These results show that the conservative assumption that Java methods might return null is by far the most frequent cause of compilation errors in the explicit nulls system. Furthermore, once these errors are removed, fewer than ten errors per thousand lines of code remain in all programs except the Scala standard library. This is quite a small number, and we consider it reasonable to expect that Scala programmers can fix the remaining errors by hand.

5.2 Evaluation of Flow-sensitive typing

In this section, we evaluate the usefulness of the flow-sensitive typing design that was described in Section 4. We have turned off flow-sensitive typing, so that each variable has a single type everywhere it is in scope, independent of any nullness tests, and again count the number of compilation errors. The error frequency with flow-sensitive typing turned off is shown in the last column of Table 2, **Ann. No Flow**. The configuration uses the annotations from the Checker Framework to specify which methods in the Java standard library return non-null values; therefore, this column is directly comparable to the **Annotations** column. The **Annotations** configuration was selected as the most precise variant of typing calls to Java methods that is still sound (assuming the Checker Framework annotations are correct).

Flow-sensitive typing makes a difference in three of the benchmarks, `stdLib213`, `scala.xml`, and `effpi`, and even there, the difference is small relative to the total number of errors. One possible reason that flow-sensitive typing has such a small impact could be that our specific flow-sensitive analysis is not sufficiently precise, so compilation errors are reported even in code that tests that references are not null; however, we will see in the next section that this is not the case. Examining the code of the community build programs, we observe that Scala programs rarely expect to encounter `null` references and thus rarely test for them, and when they do, they often use a different idiom than a test of the form `if(x != null) ...`. Specifically, many of the programs pass possibly `null` values to the constructor of the `Option` class, which turns a `null` value into the `None` object and a non-null value into an instance of `Some`. This common idiom does not require flow-sensitive typing to ensure safety.

5.3 Evaluation of other causes of nullness errors

The error counts in the **Non-null Ret.** column, where we assume that Java methods never return `null`, are low enough that it is quite feasible to manually fix the programs to remove the compilation errors. We have done this for all the programs except `stdLib213` and classified the individual causes of each compilation error. We exclude `stdLib213` not

■ **Table 3** Error classification. Libraries were migrated under **Non-null Ret.** configuration. Normalized count is in errors per thousand LOC.

Error Category	Total Count	Count per 1000 LOC
Declaration of nullable field or local symbol	74	0.81
Use of nullable field or local symbol (<code>.nn</code>)	52	0.57
Overriding error due to nullability	46	0.5
Generic <i>received</i> from Java with nullable inner type	19	0.6
Generic <i>passed</i> to Java requires nullable inner type	6	0.07
Incorrect Scala standard library definition	4	0.04
Limitation of flow typing	1	0.01
Total	202	2.21

	Modified	Total	%
LOC	484	91,337	0.53
Files	88	958	9.19

only because it has the highest error rate per thousand lines of code, but also because with 31,723 lines of code, it also has a high absolute number of errors. This analysis enables us to determine the common causes of the remaining compilation errors.

The number of errors in each category is shown in Table 3. We now explain the categories.

- *Declaration of nullable field or local symbol.* These are cases where the Scala code declares a `var` or `val` (as a field, or locally within a method) that is provably nullable because the code explicitly assigns `null` to it. For example, we might have a class field that is immediately initialized to `null`. The fix for this error is to change the type to a nullable type to reflect that the variable does (sometimes) contain a `null` reference.
- *Use of nullable field or local symbol (`.nn`).* This is the dual of the previous category. After we change the type of a variable that is sometimes null to a nullable type, all uses of that variable become nullable. Each existing use of that variable in a context that requires a non-null value then results in a compilation error, since the variable could be null. The fix for this error is to dynamically check and cast away the nullability using `.nn`.
- *Overriding error due to nullability.* This error happens when a Scala class overrides a Java-defined method that takes a reference type as an argument. Because nullification makes the argument types of the overridden method nullable, the argument types in the overriding method must also be made nullable to match the signature of the overridden method.
- *Generic received from Java with nullable inner type.* Sometimes we encounter a Java method that returns a generic with a nullified inner type. The common example are Java methods returning arrays of reference types. For example, the `split` method of the `String` class returns an `Array[String]`, which is nullified to `Array[String|Null]|Null`. This, in turn, leads to errors in Scala code that reads elements of this array and expects them to be non-null.
- *Generic passed to Java requires nullable inner type.* This happens when a Java method expects as argument a generic of some reference type (usually an `Array`). We fix these errors using `asInstanceOf` casts. An improvement to the type inference algorithm that was added to the Dotty compiler after our evaluation fixes most of these errors.⁶

⁶ <https://github.com/lampepfl/dotty/pull/8635>

- *Incorrect Scala standard library definition.* This class contains type errors that could be prevented by modifying some definition in the Scala standard library to use a more precise type. For example, the `Option.apply` method is parameterized by a type `T`, takes an argument of type `T`, and returns a value of type `Option[T]`. If the argument is null, it returns `None`; otherwise, it returns the argument wrapped in `Some`, but it never returns `Some(null)`. When this method is called on a nullable argument, for example of type `String|Null`, its return type is `Option[String|Null]`, but a more precise return type would be `Option[String]`. These errors could be fixed by future versions of the Scala standard library.
- *Limitation of flow typing.* These are cases where our implementation of flow-sensitive typing is not precise enough to model the null checks that occur in the program and prove that a value cannot be `null`. We only found one error in this class, which is due to an undiagnosed bug in our implementation that is not yet fixed.

5.4 Summary

Our results confirm the common belief that `null` references are used rarely in Scala code except for interaction with Java. For the uses of `null` that are unrelated to Java, our system reports very few compilation errors, and few changes were needed to make the community build programs compile with the explicit nulls system.

However, a large number of nullness errors are caused by values returned from calls to Java methods. Scala programmers have several options for handling these return values. The first option is to harden Scala programs to always expect and handle possible `null` references returned from Java methods. The second option is to annotate Java methods known to never return `null` references. Both these options require a significant effort. On the other hand, a third option, which requires minimal effort, is the optimistic assumption that most Java method calls will not return `null`. This option is the status quo, the state of the existing Scala code, which is not required by the compiler to explicitly consider the possibility of `null` values. There is no free lunch: there are many places in Scala code where a Java method could return `null`; one either makes the considerable effort to check and annotate or harden each such place, or one accepts the risk of a `null` reference occurring at one of those places at run time.

6 Denotational Semantics of Nullification

Type nullification is the key component that interfaces Java’s type system, where `null` is implicit, and Scala’s type system, where `null` is explicit. In this section, we give a theoretical foundation for nullification using denotational semantics. Specifically, we present λ_j and λ_s , two type systems based on a variant of System F_ω restricted to second-order type operators. In λ_j , nullability is implicit, as in Java. By contrast, in λ_s nullability is explicit, like in Scala. Nullification can then be formalized as a function that maps λ_j types to λ_s types. Following a denotational approach, we give a set-theoretic model of λ_j and λ_s . We then prove a soundness theorem stating that the meaning of types is largely unchanged by nullification.

We choose System F_ω as the basis for our formalization, rather than object-oriented calculi such as DOT [2, 27, 25] or Featherweight Generic Java [16], because type application is the challenging case for nullification. Since nullification turns Java types into Scala types, it does not need to handle many Scala-specific types (e.g. path-dependent types), so DOT is not needed for the formalization. Similarly, Featherweight Generic Java has features like inheritance that do not interact with nullification.

$S, T ::=$	λ_j Types	$\sigma, \tau ::=$	λ_s Types
int_j	int	Null	null
String _{j}	string	int_s	int
$S \times_j T$	product	String _{s}	string
$S \rightarrow_j T$	function	$\sigma + \tau$	union
$\Pi_j(X :: *_n).S$	generic	$\sigma \times_s \tau$	product
$\text{App}_j(S, T)$	type application	$\sigma \rightarrow_s \tau$	function
X	type variable	$\Pi_s(X :: *) . \sigma$	generic
		$\text{App}_s(\sigma, \tau)$	type application
		X	type variable

■ **Figure 4** Types of λ_j and λ_s . Differences are highlighted.

6.1 System F_ω , λ_j , and λ_s

We will model the Java and Scala type systems as variants of System F_ω [14, 26], the higher-order polymorphic lambda calculus. System F_ω supports universal quantification on types: e.g. we can type the (polymorphic) identity function as $\Pi X. X \rightarrow X$. The variant that we use has *second-order type operators*, which means that in the type operator $\Pi X.S$, X ranges over all types that are *not themselves type operators*. By contrast, in the unrestricted version of the calculus, X can range over other type operators. By restricting type operators, we incur a loss of expressivity: notably, we can no longer typecheck recursive data structures (which are ubiquitous in both Java and Scala). On the other hand, giving a denotational semantics for the restricted variant is much easier, because one can use a naive set-based model. More importantly, the main difficulty in designing nullification was handling Java generics. Given a generic such as `List<T>`, Java only allows instantiations of `List` with a *reference type that is not itself generic*. For example, `List<String>` is a valid type application, but `List<List>` is not. This is precisely the kind of restriction imposed by our version of System F_ω .

That said, System F_ω is too spartan: it does not distinguish between value and reference types, does not have records (present in both Java and Scala), and does not have union types (needed for explicit nulls). To remedy this we can come up with slight variations of System F_ω that have the above-mentioned features. We call these λ_j (“lambda j”) and λ_s (“lambda s”), and they are intended to stand for the Java and Scala type systems, respectively. Figure 4 shows the types of these two calculi. From now on we will focus solely on the types and will forget about terms, because nullification is a function from types to types.

λ_j extends System F_ω with integers, strings, and products (which stand in for objects). Type applications are written $\text{App}_j(S, T)$.

λ_s differs from λ_j by adding a `Null` type, type unions (written $\sigma + \tau$), and by making types be explicitly nullable, just like our version of Scala. Explicit nullability is indicated via *kinds*, as explained below.

6.1.1 Kinding Rules

In subsequent sections, we will assign meaning to types. However, we can only interpret types that are *well-kinded*. Intuitively, we need a way to differentiate between a type like $\Pi_j(X :: *_n).X$, where all variables are bound, from $\Pi_j(X :: *_n).Y$, where Y is free and so cannot be assigned a meaning.

The *kinding* rules in Figure 5 fulfill precisely this purpose. The judgment $\Gamma \vdash_j T :: K$ (resp. $\Gamma \vdash_s \sigma :: K$) establishes that type T has kind K under context Γ , and is thus well-kinded in λ_j (resp. λ_s). The different kinds K describe: nullable types ($*_n$), non-nullable types ($*_v$),

$\Gamma \vdash_j S :: K$	$\Gamma \vdash_s \sigma :: K$
$\Gamma \vdash_j \text{int}_j :: *_{\nu}$ (KJ-INT)	$\Gamma \vdash_s \text{int}_s :: *_{\nu}$ (KS-INT)
$\Gamma \vdash_j \text{String}_j :: *_{\nu}$ (KJ-STRING)	$\Gamma \vdash_s \text{String}_s :: *_{\nu}$ (KS-STRING)
$\frac{\Gamma \vdash_j S :: * \quad \Gamma \vdash_j T :: *}{\Gamma \vdash_j S \times_j T :: *_{\nu}}$ (KJ-PROD)	$\Gamma \vdash_s \text{Null} :: *_{\nu}$ (KS-NULLTYPE)
$\frac{\Gamma \vdash_j S :: * \quad \Gamma \vdash_j T :: *}{\Gamma \vdash_j S \rightarrow_j T :: *_{\nu}}$ (KJ-FUN)	$\frac{\Gamma \vdash_s \sigma :: K_1 \quad \Gamma \vdash_s \tau :: K_2 \quad K_1, K_2 \in \{*_{\nu}, *_{\nu}, *\}}{\Gamma \vdash_s \sigma + \tau :: K_1 \oplus K_2}$ (KS-UNION)
$\frac{\Gamma, X :: *_{\nu} \vdash_j S :: K}{\Gamma \vdash_j \Pi_j(X :: *_{\nu}).S :: *_{\nu} \Rightarrow K}$ (KJ-PI)	where $K \oplus K = K$, $K_1 \oplus K_2 = K_2 \oplus K_1$, $K \oplus *_{\nu} = *_{\nu}$, and $*_{\nu} \oplus * = *$
$\frac{\Gamma \vdash_j S :: *_{\nu} \Rightarrow K \quad \Gamma \vdash_j T :: *_{\nu}}{\Gamma \vdash_j \text{App}_j(S, T) :: K}$ (KJ-APP)	$\frac{\Gamma \vdash_s \sigma :: * \quad \Gamma \vdash_s \tau :: *}{\Gamma \vdash_s \sigma \times_s \tau :: *_{\nu}}$ (KS-PROD)
$\frac{\Gamma(X) = *_{\nu} \quad \Gamma \vdash_j S :: *_{\nu} \quad \Gamma \vdash_j S :: *_{\nu}}{\Gamma \vdash_j X :: *_{\nu} \quad \Gamma \vdash_j S :: * \quad \Gamma \vdash_j S :: *}$ (KJ-VAR) (KJ-NUL) (KJ-NONNULL)	$\frac{\Gamma \vdash_s \sigma :: * \quad \Gamma \vdash_s \tau :: *}{\Gamma \vdash_s \sigma \rightarrow_s \tau :: *_{\nu}}$ (KS-FUN)
$K ::=$ Kinds $*_{\nu}$ kind of nullable types $*_{\nu}$ kind of non-nullable types $*$ kind of proper types $*_{\nu} \Rightarrow K$ kind of type operators (λ_j) $* \Rightarrow K$ kind of type operators (λ_s)	$\frac{\Gamma, X :: * \vdash_s \sigma :: K}{\Gamma \vdash_s \Pi_s(X :: *).\sigma :: * \Rightarrow K}$ (KS-PI)
$\Gamma ::=$ Contexts \emptyset empty context $\Gamma, X :: *_{\nu}$ nullable type binding	$\frac{\Gamma \vdash_s \sigma :: * \Rightarrow K \quad \Gamma \vdash_s \tau :: *}{\Gamma \vdash_s \text{App}_s(\sigma, \tau) :: K}$ (KS-APP)
	$\frac{\Gamma(X) = * \quad \Gamma \vdash_s \sigma :: *_{\nu} \quad \Gamma \vdash_s \sigma :: *_{\nu}}{\Gamma \vdash_s X :: * \quad \Gamma \vdash_s \sigma :: * \quad \Gamma \vdash_s \sigma :: *}$ (KS-VAR) (KS-NUL) (KS-NONNULL)

■ **Figure 5** Kinding rules of λ_j and λ_s . Differences are highlighted.

proper (non-generic) types ($*$), and *type operators* (generics). In λ_j , type operators have kinds of the form $*_{\nu} \Rightarrow K$, modelling the fact that type arguments in Java *must* be reference types (e.g. `List<boolean>` is not well-kinded). By contrast, in λ_s (and in Scala), generics can also take value types as arguments (e.g. `List[boolean]`), so type operators have kinds of the form $* \Rightarrow K$.

The second role of the kind system is to track the *nullability* of types. Here, the difference between λ_j and λ_s is witnessed, for instance, by the KS-String rule: while in λ_j , strings are nullable ($\vdash_j \text{String}_j :: *_{\nu}$), strings in λ_s are non-nullable ($\vdash_s \text{String}_s :: *_{\nu}$). In λ_s , like in Scala, nullability can be recovered via type unions: e.g. $\vdash_s \text{String}_s + \text{Null} :: *_{\nu}$.

The rule KS-Union computes the kind of type unions. If either σ or τ contains the null value, then their union $\sigma + \tau$ also contains the null value, so $K \oplus *_{\nu} = *_{\nu}$. If one of σ or τ definitely does not contain the null value (i.e., is of kind $*_{\nu}$) and the other may or may not contain the null value (i.e., is of kind $*$), then their union $\sigma + \tau$ also may or may not contain the null value, so $*_{\nu} \oplus * = *$.

Two other rules that we want to highlight are KJ-Null and KJ-NonNull (and their λ_s counterparts). These rules give us a limited form of “subkinding”, so that $\vdash_j T :: *_v$ or $\vdash_j T :: *_n$ imply $\vdash_j T :: *$.

► **Definition 6.1** (Base kinds). *We say K is a base kind if $K \in \{*, *_n, *_v\}$.*

6.2 Denotational Semantics

Before we can prove properties of nullification, we need a *semantics* for our types and kinds. That is, so far, types and kinds are just syntactic objects, and kinding rules are syntactic rules devoid of meaning. For this task of assigning meaning we turn to the machinery of *denotational semantics*. The technical presentation is based on the treatment of predicative System F in Mitchell [20].

Here is a summary of the rest of this section. First, we construct set-theoretic models for both calculi. In this case, a model is just a family of sets that contains denotations of types and kinds. We then show how to map kinds and types to their denotations in the model. The mapping is roughly as follows: kinds \longrightarrow families of sets, proper types \longrightarrow sets, and generic types \longrightarrow functions from sets to sets. Finally, we prove a *soundness* lemma for kinding rules that says that if a type is well-kinded, then its denotation is defined and, further, it is contained in the denotation of the corresponding kind: i.e. $\Gamma \vdash_j T :: K \implies \llbracket T \rrbracket_j \eta \in \llbracket K \rrbracket_j$. The proofs of all results in this section can be found in the first author’s thesis [22].

6.2.1 Semantic Model

► **Definition 6.2** (String literals). *strings denotes the set of finite-length strings.*

The model for λ_j is a pair $\mathcal{J} = (U_1, U_2)$ of *universes* (families of sets).

U_1 is the universe of proper types. It is the least set containing $\{\mathbf{null}\}, \mathbb{Z}$, and **strings** that is closed under union, product, and functions (i.e. if u and v are in U_1 , then the set of all functions between u and v , written u^v , is also in U_1). Additionally, we define two families of sets that contain nullable and non-nullable types, respectively: $U_1^{\mathbf{null}} = \{u \mid u \in U_1, \mathbf{null} \in u\}$, and $U_1^{\mathbf{val}} = \{u \mid u \in U_1, \mathbf{null} \notin u\}$. Notice that both $U_1^{\mathbf{null}}$ and $U_1^{\mathbf{val}}$ are subsets of U_1 , and that $U_1 = U_1^{\mathbf{null}} \cup U_1^{\mathbf{val}}$.

The universe U_2 is a superset of U_1 that, additionally, contains all generic types. First, we define a family of sets $\{U_2^i\}$, for $i \geq 0$: $U_2^0 = U_1$, and $U_2^{i+1} = U_2^i \cup \{f : U_1^{\mathbf{null}} \rightarrow U_2^i\}$. Then we set $U_2 = \bigcup_{i \geq 0} U_2^i$.

The model for λ_s is very similar to the previous one. It is a pair $\mathcal{S} = (U_1, U'_2)$, where U_1 is as defined before. U'_2 is almost the same as U_2 , except that we set $U_2'^{i+1} = U_2'^i \cup \{f : U_1 \rightarrow U_2'^i\}$. Highlighted is the fact that generics in λ_s take arguments from U_1 , as opposed to $U_1^{\mathbf{null}}$.

6.2.2 Meaning of Kinds

► **Definition 6.3** (Number of arrows in a kind). *Let K be a kind. Then $\text{arr}(K)$ denotes the number of arrows (\Rightarrow) in K .*

► **Definition 6.4** (Meaning of kinds). *We give meaning to λ_j and λ_s kinds via functions $\llbracket _ \rrbracket_j$ and $\llbracket _ \rrbracket_s$, respectively. These functions are inductively defined on the structure of a kind K .*

	λ_j		λ_s
$\llbracket \text{int}_j \rrbracket_j \eta$	$= \mathbb{Z}$	$\llbracket \text{Null} \rrbracket_s \eta$	$= \{\text{null}\}$
$\llbracket \text{String}_j \rrbracket_j \eta$	$= \{\text{null}\} \cup \text{strings}$	$\llbracket \text{int}_s \rrbracket_s \eta$	$= \mathbb{Z}$
$\llbracket S \times_j T \rrbracket_j \eta$	$= \{\text{null}\} \cup (\llbracket S \rrbracket_j \eta \times \llbracket T \rrbracket_j \eta)$	$\llbracket \text{String}_s \rrbracket_s \eta$	$= \text{strings}$
$\llbracket S \rightarrow_j T \rrbracket_j \eta$	$= \llbracket S \rrbracket_j \eta^{\llbracket T \rrbracket_j \eta}$	$\llbracket \sigma \times_s \tau \rrbracket_s \eta$	$= \llbracket \sigma \rrbracket_s \eta \times \llbracket \tau \rrbracket_s \eta$
$\llbracket \Pi_j(X :: *_n).S \rrbracket_j \eta$	$= \lambda(a \in U_1^{\text{null}}). \llbracket S \rrbracket_j(\eta[X \rightarrow a])$	$\llbracket \sigma \rightarrow_s \tau \rrbracket_s \eta$	$= \llbracket \sigma \rrbracket_s \eta^{\llbracket \tau \rrbracket_s \eta}$
$\llbracket \text{App}_j(S, T) \rrbracket_j \eta$	$= \llbracket S \rrbracket_j \eta(\llbracket T \rrbracket_j \eta)$	$\llbracket \Pi_s(X :: *).\sigma \rrbracket_s \eta$	$= \lambda(a \in U_1). \llbracket \sigma \rrbracket_s(\eta[X \rightarrow a])$
$\llbracket X \rrbracket_j \eta$	$= \eta(X)$	$\llbracket \text{App}_s(\sigma, \tau) \rrbracket_s \eta$	$= \llbracket \sigma \rrbracket_s \eta(\llbracket \tau \rrbracket_s \eta)$
		$\llbracket \sigma + \tau \rrbracket_s \eta$	$= \llbracket \sigma \rrbracket_s \eta \cup \llbracket \tau \rrbracket_s \eta$
		$\llbracket X \rrbracket_s \eta$	$= \eta(X)$

■ **Figure 6** Type denotations for λ_j and λ_s . Differences are highlighted.

$$\begin{array}{ll}
\llbracket *_n \rrbracket_j = U_1^{\text{null}} & \llbracket *_n \rrbracket_s = U_1^{\text{null}} \\
\llbracket *_v \rrbracket_j = U_1^{\text{val}} & \llbracket *_v \rrbracket_s = U_1^{\text{val}} \\
\llbracket * \rrbracket_j = U_1 & \llbracket * \rrbracket_s = U_1 \\
\llbracket *_n \Rightarrow K \rrbracket_j = \{f : U_1^{\text{null}} \rightarrow \llbracket K \rrbracket_j\} & \llbracket * \Rightarrow K \rrbracket_s = \{f : U_1 \rightarrow \llbracket K \rrbracket_s\}
\end{array}$$

We can show that the meaning of kinds is contained within the corresponding model.

► **Lemma 6.5** (Kinds are well-defined). *If K is a λ_j kind, then $\llbracket K \rrbracket_j \subseteq U_2^{\text{arr}(K)}$. If K is a λ_s kind, then $\llbracket K \rrbracket_s \subseteq U_2^{\text{arr}(K)}$.*

6.2.3 Meaning of types

We now give denotations for types. To handle types that are not closed, we make the denotation functions take *two* arguments: the type whose meaning is being computed and an *environment* that gives meaning to the free variables. Additionally, we make the simplifying assumption that types have been alpha-renamed so that there are no name collisions.

► **Definition 6.6** (λ_j Environments). *A λ_j environment $\eta : \text{Var} \rightarrow U_1^{\text{null}}$ is a map from variables to elements of U_1^{null} . The empty environment is denoted by \emptyset . An environment can be extended with the notation $\eta[X \rightarrow a]$, provided that X was not already in the domain.*

► **Definition 6.7** (λ_s Environment). *A λ_s environment $\eta : \text{Var} \rightarrow U_1$ is a map from variables to elements of U_1 .*

► **Definition 6.8** (Environment Conformance). *An environment η (from λ_j or λ_s) conforms to a context Γ , written $\eta \models \Gamma$, if $\text{dom}(\eta) = \text{dom}(\Gamma)$.*

► **Definition 6.9** (Meaning of types). *We define the meaning of types via functions $\llbracket _ \rrbracket_j : \text{Types}_{\lambda_j} \rightarrow \text{Env}_{\lambda_j} \rightarrow U_2$ and $\llbracket _ \rrbracket_s : \text{Types}_{\lambda_s} \rightarrow \text{Env}_{\lambda_s} \rightarrow U_2'$. These are shown in Figure 6.*

► **Example 6.10.**

$$\begin{aligned}
\llbracket \Pi_j(X :: *_n).X \rrbracket_j \emptyset &= \lambda(a \in U_1^{\text{null}}). \llbracket X \rrbracket_j \emptyset[X \rightarrow a] \\
&= \lambda(a \in U_1^{\text{null}}). \emptyset[X \rightarrow a](X) \\
&= \lambda(a \in U_1^{\text{null}}). a \\
&= \text{id}
\end{aligned}$$

That is, the denotation of $\Pi_j(X :: *_n).X$ is the identity function that maps sets (types) in U_1^{null} to themselves.

The following lemma says that the kinding rules correctly assign kinds to our types.

► **Lemma 6.11** (Soundness of kinding rules). *The following hold:*

- $\Gamma \vdash_j T :: K$ and $\eta \vDash \Gamma$ implies $\llbracket T \rrbracket_j \eta \in \llbracket K \rrbracket_j$
- $\Gamma \vdash_s T :: K$ and $\eta \vDash \Gamma$ implies $\llbracket T \rrbracket_s \eta \in \llbracket K \rrbracket_s$

6.3 Type Nullification

Now that we have formal definitions for both λ_j and λ_s , we can also formally define type nullification. Recall that type nullification makes nullability *explicit* as we go from a type system where `null` is implicit (λ_j 's) to one where `null` is explicit (λ_s 's). For example, $(\text{int}_j) \times_j (\text{String}_j \rightarrow_j \text{String}_j)$ becomes $(\text{int}_s) \times_s (\text{String}_s + \text{Null} \rightarrow_s \text{String}_s + \text{Null})$.

That is, *type nullification is a function that turns λ_j types into λ_s types*. In the implementation (described in Section 3.1), we decided *not to nullify* arguments in type applications. That is, given a Java class `List<T>`, type applications such as `List<String>` are translated as `List<String>`, and not as `List<String|Null>`. The motivation for special casing type arguments is maximizing backwards-compatibility. Because of the different treatment for types based on whether they are in an argument position or not, we will model nullification as a *pair* of functions $(F_{\text{null}}, A_{\text{null}})$. These are defined below.

► **Definition 6.12** (Type nullification).

$$\begin{array}{ll}
 F_{\text{null}}(\text{int}_j) & = \text{int}_s & A_{\text{null}}(\text{int}_j) & = \text{int}_s \\
 F_{\text{null}}(\text{String}_j) & = \text{String}_s + \text{Null} & A_{\text{null}}(\text{String}_j) & = \text{String}_s \\
 F_{\text{null}}(X) & = X + \text{Null} & A_{\text{null}}(S \times_j T) & = F_{\text{null}}(S) \times_s F_{\text{null}}(T) \\
 F_{\text{null}}(S \rightarrow_j T) & = F_{\text{null}}(S) \rightarrow_s F_{\text{null}}(T) & A_{\text{null}}(S \rightarrow_j T) & = F_{\text{null}}(S) \rightarrow_s F_{\text{null}}(T) \\
 F_{\text{null}}(\Pi_j(X :: *_n).S) & = \Pi_s(X :: *.F_{\text{null}}(S)) & A_{\text{null}}(\text{App}_j(S, T)) & = \text{App}_s(F_{\text{null}}(S), A_{\text{null}}(T)) \\
 F_{\text{null}}(\text{App}_j(S, T)) & = \text{App}_s(F_{\text{null}}(S), A_{\text{null}}(T)) & A_{\text{null}}(X) & = X \\
 F_{\text{null}}(S \times_j T) & = (F_{\text{null}}(S) \times_s F_{\text{null}}(T)) + \text{Null} & &
 \end{array}$$

As the name suggests, A_{null} handles types that are arguments to type application, and F_{null} handles the rest. A_{null} differs from F_{null} in that it does not nullify types at the outermost level (see e.g. the `Stringj` case).

► **Definition 6.13** (Context nullification). *We lift nullification to work on contexts, turning λ_j contexts into (syntactic) λ_s contexts.*

$$\begin{array}{l}
 F_{\text{null}}(\emptyset) = \emptyset \\
 F_{\text{null}}(\Gamma, X :: *_n) = F_{\text{null}}(\Gamma), X :: *
 \end{array}$$

► **Definition 6.14** (Kind nullification). *We also lift nullification to work on kinds, turning λ_j kinds into λ_s kinds.*

$$\begin{array}{ll}
 F_{\text{null}}(K) = K & \text{if } K \text{ is a base kind} \\
 F_{\text{null}}(*_n \Rightarrow K') = * \Rightarrow F_{\text{null}}(K') & \text{otherwise}
 \end{array}$$

6.4 Soundness

We can finally prove a *soundness* result for type nullification. But what should soundness *mean* in this case? One plausible, but as it turns out, *incorrect*, definition is that nullification leaves the meaning of types *unchanged*.

► **Conjecture 6.15** (Soundness – Incorrect). *Let $\Gamma \vdash_j T :: K$, and let η be an environment such that $\eta \vDash \Gamma$. Then $\llbracket T \rrbracket_j \eta = \llbracket F_{\text{null}}(T) \rrbracket_s \eta$.*

25:20 Scala with Explicit Nulls

This conjecture is false because the meaning of generics differs between λ_j and λ_s . In both cases, generics are denoted by functions on types, but the *domains of the functions are different*:

- $\llbracket \Pi_j(X :: *_n).S \rrbracket_j \eta = \lambda(a \in U_1^{\text{null}}). \llbracket S \rrbracket_j(\eta[X \rightarrow a])$
- $\llbracket \Pi_s(X :: *) .S \rrbracket_s \eta = \lambda(a \in U_1). \llbracket S \rrbracket_s(\eta[X \rightarrow a])$

That is, λ_j generics take arguments that are in U_1^{null} (nullable arguments) and λ_s generics have wider domains and take arguments from *all* of U_1 . This matches the behaviour in Java and Scala, where the generic class `List<A>` gets “mapped” by nullification to the Scala class `List[A]`. `List<int>` is then *not valid* in Java (because `int` is not a nullable type), but `List<int>` *is* valid in Scala.

We can recover soundness via the following observation. If G is a Java generic, even though $\llbracket G \rrbracket_j$ and $\llbracket G \rrbracket_s$ are not equal, for any *valid* type application $G\langle T \rangle$ in Java, $\llbracket G\langle T \rangle \rrbracket_j = \llbracket F_{\text{null}}(G\langle T \rangle) \rrbracket_s$. That is, our soundness theorem will say that nullification leaves *fully-applied* generic types unchanged. This is just as well because *users can only manipulate values of type $G\langle T \rangle$ and never values of type G directly*.

Before we state the soundness theorem we need a few ancillary definitions.

► **Definition 6.16** (Similar Types). *Let $S, T \in U_1$. Then we say S is similar to T , written $S \sim T$, if $S \cup \{\text{null}\} = T \cup \{\text{null}\}$.*

That is, two types denotations are *similar* if they contain the same elements, except for possibly `null`. Note that \sim is symmetric.

► **Definition 6.17** (Similar Type Vectors). *Let $\vec{S} = (S_1, \dots, S_n)$ and $\vec{T} = (T_1, \dots, T_n)$ be vectors of types, where \vec{S} and \vec{T} have the same number of elements. Then $\vec{S} \sim \vec{T}$ if they are similar at every component.*

► **Definition 6.18** (Similar Environments). *Let η, η' be environments (either from λ_j or λ_s). Then η is similar to η' , written $\eta \sim \eta'$, if $\text{dom}(\eta) = \text{dom}(\eta')$ and for all type variables X in the domain, we have $\eta(X) \sim \eta'(X)$.*

Note this relation is also symmetric.

► **Definition 6.19** (Similar Kinds). *Let K_1 and K_2 be two base kinds. Then $K_1 \rightsquigarrow K_2$ is defined by case analysis.*

$$\begin{array}{ll}
 * \rightsquigarrow * & \text{(SK-PROP)} \\
 *_v \rightsquigarrow *_v & \text{(SK-NONNULL)} \\
 *_n \rightsquigarrow *_n & \text{(SK-NULL1)} \\
 *_n \rightsquigarrow *_v & \text{(SK-NULL2)} \\
 *_n \rightsquigarrow * & \text{(SK-NULL3)}
 \end{array}$$

The rules in Definition 6.19 capture what happens to the kind of a type after being transformed by A_{null} . For example, `Stringj` has kind $*_n$ in λ_j , but $A_{\text{null}}(\text{String}_j) = \text{String}_s$ has kind $*_v$ in λ_s . This is described by rule (SK-Null2). The \rightsquigarrow relation is not symmetric. This reflects the fact that A_{null} turns $*_v$ types into $*_v$ types, but can turn a $*_n$ type into a $*_v$ type.

► **Lemma 6.20.** *If K is a base kind, then $K \rightsquigarrow K$.*

Before proving soundness, we need to prove a weaker lemma that says that nullification preserves well-kindedness. This lemma is necessary because if T is well-kinded and nullification turns T into $F_{\text{null}}(T)$, the latter must be well-kinded as well.

► **Lemma 6.21** (Nullification preserves well-kindedness). *Let $\Gamma \vdash_j T :: K$ and $\Gamma' = F_{\text{null}}(\Gamma)$. Then*

1. $\Gamma' \vdash_s F_{\text{null}}(T) :: F_{\text{null}}(K)$.
2. If K is a base kind, there exists a kind K' with $K \rightsquigarrow K'$ such that $\Gamma' \vdash_s A_{\text{null}}(T) :: K'$.

► **Definition 6.22** (Curried type application). *If f is a function of m arguments and $\vec{x} = (x_1, \dots, x_m)$, we use the notation $f(\vec{x})$ to mean the curried function application $f(x_1)(x_2) \dots (x_m)$. In the degenerate case where f is not a function (i.e. $m = 0$), we set $f(\vec{x}) = f$.*

We can finally show soundness. We need to strengthen the induction hypothesis to talk about both F_{null} and A_{null} .

► **Theorem 6.23** (Soundness of type nullification). *Let $\Gamma \vdash_j T :: K$. Let η, η' be environments such that $\eta \models \Gamma$ and $\eta \sim \eta'$. Then the following two hold:*

1. If K is a base kind, then
 - a. $\llbracket T \rrbracket_j \eta = \llbracket F_{\text{null}}(T) \rrbracket_s \eta'$ and
 - b. $\llbracket T \rrbracket_j \eta \sim \llbracket A_{\text{null}}(T) \rrbracket_s \eta'$.
2. If K is a type application with $\text{arr}(K) = m$, let \vec{x} and \vec{y} be two m -vectors of elements of U_1^{null} and U_1 , respectively, with $\vec{x} \sim \vec{y}$. Then $\llbracket T \rrbracket_j \eta(\vec{x}) = \llbracket F_{\text{null}}(T) \rrbracket_s \eta'(\vec{y})$.

The first assertion in the soundness theorem says that the meaning of base (non-generic) types is unchanged by nullification. For example, the denotations of String_j and $F_{\text{null}}(\text{String}_j) = \text{String}_s + \text{Null}$ are equal. The second assertion says that if we start with a generic type that takes K arguments and apply it *fully* (i.e. apply it to K arguments), and *then* we apply nullification, the meaning of the type application is also unchanged, provided that the original type application is well-kinded in λ_j . For example, in λ_j we can represent generic pairs by $\text{Pair} = \Pi_j(X :: *_n). \Pi_j(Y :: *_n). X \times_j Y$. The theorem says that if $\text{Pair}_{(T_1, T_2)} = \text{App}_j(\text{App}_j(\text{Pair}, T_1), T_2)$ is well-kinded in λ_j (i.e. both T_1 and T_2 are in $*_n$), then the meaning of $\text{Pair}_{(T_1, T_2)}$ is also unchanged by nullification. That is, $\llbracket \text{Pair}_{(T_1, T_2)} \rrbracket_j = \llbracket F_{\text{null}}(\text{Pair}_{(T_1, T_2)}) \rrbracket_s$.

6.5 Discussion

As Section 3.1 points out, both underapproximations and overapproximations in nullification would lead to unsoundness, so “preserves elements of types” is a useful soundness criterion for type nullification.

That the meaning of types with base kinds remains unchanged is important, because program values always have base kinds. The meaning of generics *is* changed by nullification. This reflects the fact that, in λ_s and Scala, type arguments can be either value or reference types, while in λ_j and Java only reference types can be used. The soundness theorem (Theorem 6.23) in this section shows that *fully-applied* generics (which have base kinds) remain unchanged. Extrapolating, this means that Java types corresponding to fully-applied generics (e.g. `ArrayList<String>`), can be represented *exactly* in Scala. The other direction does not hold; e.g. the Scala type `List[Int]` cannot be represented directly in Java (because `Int` is a value type). Instead, `List[Int]` must be translated as `List<Integer>` or `List<Object>`, where `Integer` is the Java type for boxed integers. The type translation from Scala to Java (erasure) is not modelled in this section and remains as future work.

7 Related Work

The related work we have identified can be divided into four classes:

- Type systems for nullability in modern, widely used programming languages.
- Schemes to guarantee sound initialization. These have been mostly implemented as research prototypes, or as pluggable type systems.
- Pluggable type systems that are not part of the “core” of a programming language, but are used as checkers that provide additional guarantees (in our case, related to nullability).
- Denotations of types.

7.1 Nullability in the Mainstream

Kotlin is an object-oriented, statically-typed programming language for the JVM [17]. Kotlin’s flow typing handles both `vars` and `vals`, while our system currently only supports `vals`. Additionally, Kotlin can recognize nullability annotations not just at the top-level, but also within type arguments to generics. Nullability in Kotlin is expressed with a type modifier: the reference type `T` is non-nullable, but `T?` is nullable. By contrast, in our design explicit nullability is achieved through a combination of union types and a new type hierarchy. The two approaches are comparable in their expressiveness, but in a language with support for union types (such as Scala), our approach expresses nullability as a derived concept and avoids introducing new kinds of types.

Kotlin handles Java interoperability via *platform types*. A platform type, written `T!`, is a type with *unknown* nullability. Kotlin turns all Java-originated types into platform types. Given a type `T!`, Kotlin allows casting it (automatically) into both a `T?` and a `T`. The cast from `T!` to `T?` always succeeds, but the cast from `T!` to `T` might fail at runtime, because the Kotlin compiler automatically inserts a runtime assertion that the value being cast is non-null. We chose to represent types flowing into Scala code from Java using union types and the `JavaNull` annotation. In this way we avoid introducing a new kind of type (platform types) into the already-crowded Scala type system. Another reason for diverging from the platform types approach is soundness. Kotlin allows (unsound) member selections on platform types, just like we do in Scala via `JavaNull`, but platform types are even more permissive. For example, Kotlin automatically casts a value of platform type `String!` to the non-nullable type `String`; by contrast, in our design the type `String | JavaNull` is *not* a subtype of `String`, so the cast needs to be applied manually. We can think of platform types as a generalization of `JavaNull` that allows not only member selections, but also subtyping with respect to non-nullable types. We wanted to strike a balance between soundness and usability in our design, so we opted for a more restrictive approach than Kotlin’s in the handling of Java-originated types.

Ceylon is another object-oriented, statically-typed language that also targets the JVM [12]. Ceylon has union and intersection types, like Scala, and represents explicit nullability via union types [13]. The main difference between Ceylon’s design and ours is the handling of interoperability with Java. Ceylon, like Kotlin, takes an “optimistic” approach where all Java-originated types used in Ceylon code are assumed to be non-nullable (and checked as such at runtime, with automatically-generated assertions). By contrast, we assume that all Java reference types are nullable, and so develop a type nullification function that turns Java types, including generics, into equivalent Scala types. To our knowledge, we are the first to formally describe type nullification, as well propose and prove a correctness criteria for it (nullification preserves values of types).

Swift is a statically-typed programming language, originally designed for the iOS and macOS ecosystems [4]. Swift has a `nil` reference, which is similar to `null` in Scala [5]. Types in Swift are non-nullable by default, but one can use *optionals* to get back nullability. For example, the type `Int?`, an optional, is either an integer or `nil`. Optionals can be *force unwrapped* using the `!` operator, which potentially raises a runtime error if the underlying optional is `nil`. Swift also has a notion of *optional binding*, which is similar to the monadic `bind` operator in Haskell [30], but specialized for optionals. Additionally, Swift has *implicitly unwrapped optionals*, which are similar to Kotlin’s platform types. That is, the type `Int!`, an implicitly unwrapped optional, need not be forced unwrapped explicitly before a value can be retrieved, but if the underlying value is `nil`, it will produce a runtime error.

C# has reference types that are non-nullable by default. Compared to our design, *C#* offers more fine-grained control over where explicit nulls are enabled. In our system, explicit nulls can only be enabled or disabled at the project level. In *C#*, the user can additionally opt in to explicit nulls for *specific code regions* via “pragmas” (program metadata).

7.2 Sound Initialization

Even with a type system that has non-nullable types, there is a possibility of unsoundness because of *incomplete initialization*. This can happen, for example, due to dynamic dispatch, or *leaking* of the `this` reference from the constructor to helper methods. The problem is that in an uninitialized (or *partially* uninitialized) object, the *invariants* enforced by the type system need not hold yet. Specifically, fields that are marked as non-null might *nevertheless* be `null` (or contain nonsensical data) because they have not yet been initialized.

Over the years, many solutions have been proposed for the sound initialization problem, usually involving a combination of type system features and static analyses. These prior designs include *raw types* [10], *masked types* [24], *delayed types* [11], the *Freedom Before Commitment* scheme [29], and X10’s *hardhat* design [31]. These last two schemes have been identified as the bases for a sound initialization scheme for Scala [19].

7.3 Pluggable Type Checkers

Another line of work that is relevant to nullability is *pluggable type checkers*. A pluggable type checker is a custom-built typechecker that *refines* the typing rules of a host system [23].

The *Checker Framework* [23] is a framework for building pluggable type checkers for Java. Users have the option of writing their typecheckers in a *declarative* style, which requires less work (they do not need to write Java code) but is less expressive, or in a *procedural* style, where the checker can have arbitrarily complex logic, but is therefore harder to implement. One of the checkers that comes “pre-packaged” with the framework is the Nullness Checker. In fact, “the Nullness Checker is the largest checker by far that has been built with the Checker Framework” [9]. As of 2017, the Nullness Checker implemented a variant of the Freedom Before Commitment scheme, as well as support for flow typing and multiple heuristics to improve the accuracy of its static analysis [7, 9]. Dietl et al. [9] conducted an extensive evaluation of the Nullness Checker in production code, finding multiple errors in the Google Collections library for Java.

The *Granullar* project [7] combines the null checker from the Checker Framework with techniques from *gradual typing* [28]. Granullar allows the user to migrate only part of a project to use null checks. To that effect, the code under consideration is divided into *checked* and *unchecked* regions. Nullability checks are done statically within the checked region, using the Freedom Before Commitment scheme implemented by the Checker Framework.

No checks are done for the unchecked portion of the code. However, Granular *insulates* the checked region from unsafe interactions with the unchecked region by inserting *runtime* non-null checks at the boundary.

NullAway [6] is a nullness checker for Android applications developed at Uber. NullAway is implemented as a pluggable type system on top of the Error Prone framework [1]. NullAway trades away soundness for efficiency. Specifically, the tool is *unsound* in multiple ways: its initialization checks ignore the problem of leaking the `this` reference, all unchecked methods are assumed to return *non-null* values, and flow typing assumes that all methods are *pure* and *deterministic*. In exchange for the unsoundness, NullAway has a lower build-time (2.5x) and annotation overheads than similar tools (2.8 - 5.1x) [6]. After extensive empirical evaluation [6], NullAway’s authors note that the unsound assumptions do not lead to nullability errors in practice.

7.4 Semantics of Nullification

The model of System F_ω that we used is based on the one given by Mitchell [20] for System F (which, in turn, is based on Bruce et al. [8]). The denotations for sums and product types are standard in the literature.

There is one deviation from Mitchell [20] in how we construct denotations for generics. The standard way is to say that the denotation of a generic type is an (infinite) Cartesian product, whereas we use a simple function on types. That is, instead of saying $\llbracket \Pi_s(X :: *) . S \rrbracket_s \eta = \prod_{a \in U_1} \llbracket S \rrbracket_j (\eta[X \rightarrow a])$, we define $\llbracket \Pi_s(X :: *) . S \rrbracket_s \eta = \lambda(a \in U_1) . \llbracket S \rrbracket_s (\eta[X \rightarrow a])$. The reason for the discrepancy is that λ_j and λ_s have type applications at the type level (e.g. $\text{App}_j(S, T)$), whereas in System F, type applications are terms (e.g. $t [T]$). If we use the variant with the Cartesian product, then $\llbracket \text{App}_s(\Pi_s(X :: *) . X, \text{int}_s) \rrbracket_s \emptyset$ would be an *element* of $\llbracket \text{int}_s \rrbracket_s$ (an element of \mathbb{Z}). However, what we need for the soundness theorem is that $\llbracket \text{App}_s(\Pi_s(X :: *) . X, \text{int}_s) \rrbracket_s \emptyset$ be *equal* to $\llbracket \text{int}_s \rrbracket_s$, hence the second definition.

The novelty of our work is the use of denotational semantics for reasoning specifically about nullification. We are not aware of any related work that formalizes and proves soundness of nullification.

8 Conclusions

In this paper, we described a modification to the Scala type system that makes nullability explicit in the types. Reference types are no longer nullable, and nullability can be recovered using type unions. Because interoperability with Java is important, a type nullification phase translates Java types into Scala types. A simple form of flow typing allows for more idiomatic handling of nullable values. We implemented the design as a modification to the Dotty compiler.

To evaluate the implementation of explicit nulls, we migrated Scala programs from the Dotty community build to use the new type system. The results confirm that Scala code uses null references sparingly and that our system requires few modifications to the Scala internals of existing programs, with the significant exception of the places where Scala code interacts with Java code. The Java type system does not provide information about which references could be null. A Scala programmer faces an inevitable choice: One option is to annotate the Java code or to defensively check for the possibility of a null reference at every call to a Java method (and the type system can enforce such checks), but this requires considerable effort in programs that contain many such calls. Another option is to configure the type system to

optimistically but unsoundly assume that Java methods do not return null, which makes migration to the type system easy, but retains the possibility of null dereference exceptions if the Java methods violate the assumption.

We also showed how the intuitive reasoning about nullification based on sets can be given a solid formal footing, via denotational semantics. First, we presented λ_j and λ_s , two type systems based on System F_ω restricted to second-order type operators. These type systems formalize the implicit and explicit nature of `null` in Java and Scala, respectively. We then gave simple set-theoretic models for λ_j and λ_s , which in turn allow us to define denotations for types and kinds. We formalized nullification as a function from λ_j types to λ_s types. Finally, we proved a soundness theorem that says that nullification leaves the meaning of types largely unchanged.

References

- 1 Edward Aftandilian, Raluca Sauciu, Siddharth Priya, and Sundaresan Krishnan. Building Useful Program Analysis Tools Using an Extensible Java Compiler. In *2012 IEEE 12th International Working Conference on Source Code Analysis and Manipulation*, pages 14–23. IEEE, 2012.
- 2 Nada Amin, Samuel Grütter, Martin Odersky, Tiark Rompf, and Sandro Stucki. The essence of dependent object types. In *A List of Successes That Can Change the World - Essays Dedicated to Philip Wadler on the Occasion of His 60th Birthday*, pages 249–272, 2016.
- 3 Nada Amin and Ross Tate. Java and Scala’s Type Systems are Unsound: The Existential Crisis of Null Pointers. 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, part of SPLASH 2016, Amsterdam, The Netherlands, October 30 - November 4, 2016*, pages 838–848. ACM, 2016. doi:10.1145/2983990.2984004.
- 4 Apple Inc. About swift. [Online; accessed 5-November-2019]. URL: <https://docs.swift.org/swift-book/>.
- 5 Apple Inc. Swift language guide. [Online; accessed 5-November-2019]. URL: <https://docs.swift.org/swift-book/LanguageGuide/TheBasics.html>.
- 6 Subarno Banerjee, Lazaro Clapp, and Manu Sridharan. Nullaway: Practical Type-Based Null Safety for Java. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 740–750. ACM, 2019.
- 7 Dan Brotherston, Werner Dietl, and Ondřej Lhoták. Granular: Gradual Nullable Types for Java. In *Proceedings of the 26th International Conference on Compiler Construction*, pages 87–97. ACM, 2017.
- 8 Kim B Bruce, Albert R Meyer, and John C Mitchell. The Semantics of Second-Order Lambda Calculus. *Information and Computation*, 85(1):76–134, 1990.
- 9 Werner Dietl, Stephanie Dietzel, Michael D Ernst, Kivanç Muşlu, and Todd W Schiller. Building and Using Pluggable Type-Checkers. In *Proceedings of the 33rd International Conference on Software Engineering*, pages 681–690. ACM, 2011.
- 10 Manuel Fähndrich and K. Rustan M. Leino. Declaring and Checking Non-Null Types in an Object-Oriented Language. In Ron Crocker and Guy L. Steele Jr., editors, *Proceedings of the 2003 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications, OOPSLA 2003, October 26-30, 2003, Anaheim, CA, USA*, pages 302–312. ACM, 2003. doi:10.1145/949305.949332.
- 11 Manuel Fähndrich and Songtao Xia. Establishing Object Invariants with Delayed Types. In Richard P. Gabriel, David F. Bacon, Cristina Videira Lopes, and Guy L. Steele Jr., editors, *Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2007, October 21-25, 2007, Montreal, Quebec, Canada*, pages 337–350. ACM, 2007. doi:10.1145/1297027.1297052.

- 12 Gavin King. The Ceylon Language. [Online; accessed 30-May-2020]. URL: <https://ceylon-lang.org/>.
- 13 Gavin King. Using Java From Ceylon. [Online; accessed 30-May-2020]. URL: <https://ceylon-lang.org/documentation/1.2/reference/interoperability/java-from-ceylon/>.
- 14 Jean-Yves Girard. *Interprétation fonctionnelle et élimination des coupures de l'arithmétique d'ordre supérieur*. PhD thesis, Université Paris VII, 1972.
- 15 Arjun Guha, Claudiu Saftoiu, and Shriram Krishnamurthi. Typing Local Control and State Using Flow Analysis. In *European Symposium on Programming*, pages 256–275. Springer, 2011.
- 16 Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler. Featherweight Java: a minimal core calculus for Java and GJ. *ACM Trans. Program. Lang. Syst.*, 23(3):396–450, 2001.
- 17 Kotlin Foundation. Kotlin programming language. [Online; accessed 5-November-2019]. URL: <https://kotlinlang.org/>.
- 18 Tim Lindholm, Frank Yellin, Gilad Bracha, and Alex Buckley. *The Java virtual machine specification*. Pearson Education, 2014.
- 19 Fengyun Liu, Aggelos Biboudis, and Martin Odersky. Initialization Patterns in Dotty. In *Proceedings of the 9th ACM SIGPLAN International Symposium on Scala*, pages 51–55. ACM, 2018.
- 20 John C Mitchell. *Foundations for Programming Languages*, volume 1. MIT press Cambridge, 1996.
- 21 MITRE. 2019 CWE Top 25 Most Dangerous Software Errors. [Online; accessed 17-November-2019]. URL: https://cwe.mitre.org/top25/archive/2019/2019_cwe_top25.html.
- 22 Abel Nieto Rodriguez. Scala with explicit nulls. Master's thesis, University of Waterloo, 2019.
- 23 Matthew M Papi, Mahmood Ali, Telmo Luis Correa Jr, Jeff H Perkins, and Michael D Ernst. Practical Pluggable Types for Java. In *Proceedings of the 2008 international symposium on Software testing and analysis*, pages 201–212. ACM, 2008.
- 24 Xin Qi and Andrew C. Myers. Masked Types for Sound Object Initialization. In Zhong Shao and Benjamin C. Pierce, editors, *Proceedings of the 36th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2009, Savannah, GA, USA, January 21-23, 2009*, pages 53–65. ACM, 2009. doi:10.1145/1480881.1480890.
- 25 Marianna Rapoport, Ifaz Kabir, Paul He, and Ondřej Lhoták. A simple soundness proof for dependent object types. *PACMPL*, 1(OOPSLA):46:1–46:27, 2017.
- 26 John C Reynolds. Towards a Theory of Type Structure. In *Programming Symposium*, pages 408–425. Springer, 1974.
- 27 Tiark Rompf and Nada Amin. Type soundness for dependent object types (DOT). In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2016, part of SPLASH 2016, Amsterdam, The Netherlands, October 30 - November 4, 2016*, pages 624–641, 2016.
- 28 Jeremy G Siek and Walid Taha. Gradual Typing for Functional Languages. In *Scheme and Functional Programming Workshop*, volume 6, pages 81–92, 2006.
- 29 Alexander J. Summers and Peter Müller. Freedom Before Commitment: a Lightweight Type System for Object Initialisation. In Cristina Videira Lopes and Kathleen Fisher, editors, *Proceedings of the 26th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2011, part of SPLASH 2011, Portland, OR, USA, October 22 - 27, 2011*, pages 1013–1032. ACM, 2011. doi:10.1145/2048066.2048142.
- 30 Philip Wadler. Monads for Functional Programming. In *International School on Advanced Functional Programming*, pages 24–52. Springer, 1995.
- 31 Yoav Zibin, David Cunningham, Igor Peshansky, and Vijay Saraswat. Object Initialization in X10. In *European Conference on Object-Oriented Programming*, pages 207–231. Springer, 2012.

A Type-Directed Operational Semantics For a Calculus with a Merge Operator

Xuejing Huang 

The University of Hong Kong, China
xjhuang@cs.hku.hk

Bruno C. d. S. Oliveira

The University of Hong Kong, China
bruno@cs.hku.hk

Abstract

Calculi with disjoint intersection types and a merge operator provide general mechanisms that can subsume various other features. Such calculi can also encode highly dynamic forms of object composition, which capture common programming patterns in dynamically typed languages (such as JavaScript) in a fully statically typed manner. Unfortunately, unlike many other foundational calculi (such as *System F*, *System F_<*, or *Featherweight Java*), recent calculi with the merge operator lack a (direct) operational semantics with standard and expected properties such as *determinism* and *subject-reduction*. Furthermore the metatheory for such calculi can only account for *terminating programs*, which is a significant restriction in practice.

This paper proposes a *type-directed operational semantics* (TDOS) for λ_i^{\dagger} : a calculus with *intersection types* and a *merge operator*. The calculus is inspired by two closely related calculi by Dunfield (2014) and Oliveira et al. (2016). Although Dunfield proposes a direct small-step semantics for her calculus, her semantics lacks both *determinism* and *subject-reduction*. Using our TDOS we obtain a direct semantics for λ_i^{\dagger} that has both properties. To fully obtain determinism, the λ_i^{\dagger} calculus employs a disjointness restriction proposed in Oliveira et al.'s λ_i calculus. As an added benefit the TDOS approach deals with recursion in a straightforward way, unlike λ_i and subsequent calculi where recursion is problematic. To further relate λ_i^{\dagger} to the calculi by Dunfield and Oliveira et al. we show two results. Firstly, the semantics of λ_i^{\dagger} is sound with respect to Dunfield's small-step semantics. Secondly, we show that the type system of λ_i^{\dagger} is complete with respect to the λ_i type system. All results have been fully formalized in the Coq theorem prover.

2012 ACM Subject Classification Theory of computation → Type theory; Software and its engineering → Object oriented languages; Software and its engineering → Polymorphism

Keywords and phrases operational semantics, type systems, intersection types

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2020.26

Supplementary Material ECOOP 2020 Artifact Evaluation approved artifact available at <https://doi.org/10.4230/DARTS.6.2.9>.
<https://github.com/XSnow/ECOOP2020>

Funding This work has been sponsored by Hong Kong Research Grant Council projects number 17210617 and 17209519.

Acknowledgements We thank the anonymous reviewers for their helpful comments.

1 Introduction

The *merge operator* was firstly introduced by Reynolds in the Forsythe language [43] over 30 years ago. It has since been studied, refined and used in some language designs by multiple researchers [2, 5, 9, 20, 23, 39]. At its essence the merge operator allows creating values that can have *multiple types* (encoded as *intersection types* [18, 41]). For example, with the merge operator, the following program is valid:

$$\text{let } x : \text{Int} \& \text{Bool} = 1, , \text{True in } (x + 1, \text{not } x)$$


© Xuejing Huang and Bruno C. d. S. Oliveira;
licensed under Creative Commons License CC-BY
34th European Conference on Object-Oriented Programming (ECOOP 2020).
Editors: Robert Hirschfeld and Tobias Pape; Article No. 26; pp. 26:1–26:32



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



Here the variable x has two types, expressed by the intersection type $\text{Int} \& \text{Bool}$. The corresponding value for x is built using the merge operator $(,)$. Later uses of x , such as the expression $(x + 1, \text{not } x)$ can use x both as an integer or as a boolean. For this particular example, the result of executing the expression is the pair $(2, \text{False})$.

The merge operator adds expressive power to calculi with intersection types. Much work on intersection types has focused on *refinement intersections* [21, 24, 28], which only increase the expressive power of types. In systems with refinement intersections, types can simply be erased during compilation. However, in those systems the intersection type $\text{Int} \& \text{Bool}$ is invalid since Int and Bool are not refinements of each other. In other systems, including many OO languages with intersection types [25, 34, 36, 42], the type $\text{Int} \& \text{Bool}$ has no inhabitants and the simple program above is inexpressible. The merge operator adds expressiveness to terms and allows constructing values that inhabit the intersection type $\text{Int} \& \text{Bool}$.

There are various practical applications for the merge operator. One benefit, as Dunfield [23] argues, is that the merge operator and intersection types provide “*general mechanisms that subsume many different features*”. This is important because often a new type system feature involves extending the metatheory and implementation, which can be non-trivial. If instead we provide general mechanisms that can encode such features, then adding new features will become a lot easier. Dunfield has illustrated this point by showing that *multi-field records*, *overloading* and forms of *dynamic typing* can all be easily encoded in the presence of the merge operator. More recently, the merge operator has been used in calculi with disjoint intersection types [2, 5, 20] to encode several non-trivial object-oriented features, which enable highly dynamic forms of object composition not available in current mainstream languages such as Scala or Java. These include *first-class traits* [4], *dynamic mixins* [2], and forms of *family polymorphism* [5]. These features allow, for instance, capturing widely used and expressive techniques for object composition used by JavaScript programmers (and programmers in other dynamically typed languages), but in a completely *statically type-safe* manner [2, 4]. For example, in the SEDEL language [4], which is based on disjoint intersection types, we can define and use first-class traits such as:

```
// addId takes a trait as an argument, and returns another trait
addId(super : Trait[Person], idNumber : Int) : Trait[Student] =
  trait inherits super => { // dynamically inheriting from an unknown person
    def id : Int = idNumber
  }
```

Similarly to classes in JavaScript, first-class traits can be passed as arguments, returned as results, and they can be constructed dynamically (at run-time). In the program above inheritance is encoded as a merge in the core language with disjoint intersection types used by SEDEL.

Despite over 30 years of research, the semantics of the merge operator has proved to be quite elusive. Because of its foundational importance, we would expect a simple and clear *direct* semantics to exist for calculi with a merge operator. After all, this is what we get for other foundational calculi such as the *simply typed lambda calculus*, *System F*, *System F_ω*, the *calculus of constructions*, *System F_<*, *Featherweight Java* and others. All these calculi have a simple and elegant *direct operational semantics* (often presented in a small-step style [55]). While for the merge operator there have been efforts in the past to define a direct operational semantics, these efforts have placed severe limitations that disallow many of the applications previously discussed or they lacked important properties. Reynolds [44] was the first to look at this problem, but in his calculus the merge operator is severely limited (for instance a merge of two functions is not possible). Castagna [9] studied another calculus, where only merges of functions are possible. Pierce [39] was the first to briefly consider a calculus with

an unrestricted merge operator (called *glue* in his own work). He discussed an extension to F_{\wedge} with a merge operator but he did not study the dynamic semantics with the extension. Finally, Dunfield [23] goes further and presents a direct operational semantics for a calculus with an unrestricted merge operator. However the problem is that subject-reduction and determinism are lost.

Dunfield also presents an alternative way to give the semantics for a calculus with the merge operator *indirectly by elaboration* to another calculus. This elaboration semantics is type-safe and offers, for instance, a reasonable implementation strategy, and it is also employed in more recent work on the merge operator with disjoint intersection types. However the elaboration semantics has two important drawbacks. Firstly, reasoning about the elaboration semantics is much more complex: to understand the semantics of programs with the merge operator we have to understand the translation and semantics of the target calculus. This complicates informal and formal reasoning. Secondly, a fundamental property in an elaboration semantics is *coherence* [44] (which ensures that the meaning of a program is not ambiguous). All existing calculi with disjoint intersection types prove coherence, but this currently comes at a high price: the calculi and proof techniques employed to prove coherence can only deal with terminating programs. A severe limitation in practice!

This paper presents a *type-directed operational semantics* (TDOS) for λ_i^{\ddagger} : a calculus with intersection types and a merge operator [43]. λ_i^{\ddagger} is inspired by closely related calculi by Dunfield [23] and Oliveira et al. (λ_i) [20], but addresses two key difficulties in the dynamic semantics of calculi with a merge operator. The first one is the type-dependent nature of the merge operator. This difficulty is addressed by using types in the TDOS to guide reduction, which is crucial to prove *subject-reduction*. The second difficulty is that a fully unrestricted merge operator is inherently ambiguous. For instance the merge $1, 2$ can evaluate to both 1 and 2. Therefore some restriction is still necessary for a deterministic semantics. To fully obtain determinism, the λ_i^{\ddagger} calculus uses the disjointness restriction that is employed in λ_i and several other calculi using disjoint intersection types, and two important new notions: *typed reduction* and *consistency*. Typed reduction is a reduction relation that can further reduce values under a certain type. In other words, type annotations influence operational behavior: two programs that differ only in type annotations may behave differently. Consistency is an equivalence relation on values, that is key for the determinism result. Determinism in TDOS offers the same guarantee that coherence offers in an elaboration semantics (both properties ensure that the semantics is unambiguous), but it is much simpler to prove. Additionally, the TDOS approach deals with recursion in a straightforward way, unlike λ_i and subsequent calculi where recursion is very problematic for proving coherence.

To further relate λ_i^{\ddagger} to the calculi by Dunfield and Oliveira et al. we show two results. Firstly, we show that the type system of λ_i^{\ddagger} is complete with respect to the type system of λ_i . Secondly, the semantics of λ_i^{\ddagger} is sound with respect to Dunfield's semantics. In our work we use two variants of λ_i^{\ddagger} : one that follows Dunfield's original formulation of subtyping, and another with a more powerful subtyping relation inspired by Bi et al. [5]. The more powerful subtyping relation enables λ_i^{\ddagger} to account for merges of functions in a natural way, which was awkward in λ_i . For the variant with the extension we also require a minor extension to Dunfield's operational semantics. The two variants of the λ_i^{\ddagger} calculus and its metatheory have been fully formalized in the Coq theorem prover.

In summary, the contributions of this paper are:

- **The λ_i^{\ddagger} calculus and its TDOS:** We present a TDOS for λ_i^{\ddagger} : a calculus with intersection types and a merge operator. The semantics of λ_i^{\ddagger} is both *deterministic* and it has *subject-reduction*.

- **Support for non-terminating programs:** Our new proof methods can deal with recursion, unlike the proof methods used in previous calculi with disjoint intersection types [5, 6], due to limitations of the current proof approaches for coherence.
- **Typed reduction and consistency:** We propose the novel notions of typed reduction and consistency, which are useful to prove determinism and subject-reduction.
- **Relation with other calculus with intersection types:** We relate λ_i^{\dagger} with the calculi proposed by Dunfield and Oliveira et al (λ_i). In short all programs that are accepted in λ_i can type-check with our type system, and the semantics of λ_i^{\dagger} is sound with respect to Dunfield's semantics.
- **Coq formalization:** All the results presented in this paper have been formalized in the Coq theorem prover and they are available in the supplementary material.

2 Overview

This section gives an overview of the type-directed operational semantics for λ_i^{\dagger} . We first provide some background about the applications of the merge operator. Then we introduce Dunfield's untyped semantics [23], and identify its problems: the non-determinism of the semantics and the lack of subject-reduction. Dunfield's semantics is nonetheless used to guide the design of our own TDOS. We show how the TDOS of λ_i^{\dagger} uses type annotations to guide reduction, thus obtaining a deterministic semantics that also has the subject-reduction property.

2.1 First-Class Traits: An Application of the Merge Operator

To give an idea of the kinds of applications for calculi with a merge operator, we briefly present one existing application: *typed first-class traits* [4]. *Traits* [46] in object-oriented programming provide a model of multiple inheritance. Both traits and *mixins* [27] encapsulate a collection of related methods to be added to a class. When composing multiple traits/mixins, conflicts are dealt differently. Mixins use the order of composition to determine which implementation to pick. Traits require programmers to explicitly resolve the conflicts instead, and reject compositions with conflicts. Merges with disjoint intersection types are closely related to traits because merges with conflicts are also rejected.

Here we borrow an example from the SEDEL language [4] to demonstrate how it encodes (typed) first-class traits and dynamic inheritance via the merge operator.

```

type Editor = {on_key : String → String, do_cut : String, show_help : String};
type Version = {version : String};

trait editor [self : Editor & Version] ⇒ {
  on_key(key : String) = "Pressing " ++ key;
  do_cut = self.on_key "C-x" ++ " for cutting text";
  show_help = "Version: " ++ self.version ++ " Basic usage..."
};

```

In SEDEL traits are elaborated into a core calculus with disjoint intersection types and a merge operator. A trait can be viewed as a function taking a self argument and producing a record. In this example, the record, which contains three fields, is encoded as a merge of three single field records. Because all the fields have distinct field names, the merge is disjoint and the definition is accepted. Similarly to a JavaScript class, in the trait `editor`, the `doCut` method calls the `onKey` method via the self reference and it is dynamically dispatched. What is more, traits in SEDEL have a self type annotation which is similar to Scala [36]. In this example, the type of the self reference is the intersection of two record types `Editor` and

`Version`. Note that `show_help` is defined in terms of an undefined `version` method. Usually, in a statically typed language like Java, an abstract method is required, making `editor` an abstract class. Instead, SEDEL encodes abstract methods via self-types. The requirements stated by the type annotation of `self` must be satisfied when later composing `editor` with other traits, i.e. an implementation of the method `version` should be provided.

The interesting features in SEDEL are that traits are *first-class* and inheritance can be *dynamic*. The following example illustrates such features:

```

type Spelling = {check : String};

spell_mixin (base : Trait[Editor & Version, Editor]) =
  trait [self : Editor & Version] inherits base => {
    override on_key(key : String) = "Process " ++ key ++ " on spell editor";
    check = super.on_key "C-c" ++ " for spelling check"
  }

```

The above function takes a trait as an argument, and returns a trait as a result. The argument `base` is a trait of type `Trait[Editor & Version, Editor]`, where the two types denote trait requirements and functionality respectively. The trait `editor` has type `Trait[Editor & Version, Editor]`, since it requires `Editor & Version` and only provides those method specified by `Editor`. Therefore, `editor` can be used as an argument for `spell_mixin`. Note that unlike mainstream OOP languages like Java, the inherited trait (which would correspond to a superclass in Java) is *parametrized*, thus enabling *dynamic inheritance*. In SEDEL the choice of the inherited trait (i.e. the superclass) can happen at run-time, unlike in languages with static inheritance (such as Java or Scala). Finally, also note the use of the keyword `override` to override `on_key`. Without such keyword the definition of `spell_mixin` would be rejected due to a conflict (or a violation of disjointness), since `base` already provides an implementation of `on_key`. For a more detailed description of SEDEL and first-class traits we refer the reader to the work by Bi et al. [4].

2.2 Background: Dunfield's Non-Deterministic Semantics

Dunfield studied the semantics of a calculus with intersection types and a merge operator. The interesting aspect of her calculus is the merge operator, which takes two terms e_1 and e_2 , of some types A and B , to create a new term that can behave both as a term of type A and as a term of type B . Intersection types and the merge operator in Dunfield's calculus are similar to pair types and pairs. Indeed, a program written with pairs that behaves identically to the program shown in Section 1 is:

$$\text{let } x : (\text{Int}, \text{Bool}) = (1, \text{True}) \text{ in } (\text{fst } x + 1, \text{not } (\text{snd } x))$$

However while for pairs both the introductions and eliminations are explicit, with the merge operator the eliminations (i.e. projections) are *implicit* and driven by the types of the terms. Dunfield exploits this similarity to give a type-directed elaboration semantics to her calculus. The elaboration transforms merges into pairs, intersection types into pair types and inserts the missing projections.

Syntax. The top of Figure 1 shows the syntax of Dunfield's calculus. Types include a top type `Top`, function types ($A \rightarrow B$) and intersection types (written as $A \& B$). Most expressions are standard, except the merges ($E_1 , , E_2$). The calculus also includes a canonical top value \top , and allows variables to be values. Note that the original Dunfield's calculus uses a different notation for intersection types ($A \wedge B$), and supports union types ($A \vee B$).

<i>Type</i>	$A, B ::= \text{Top} \mid A \rightarrow B \mid A \& B$		
<i>Expr</i>	$E ::= x \mid \top \mid \lambda x. E \mid E_1 E_2 \mid \mathbf{fix} x. E \mid E_1 , , E_2$		
<i>Value</i>	$V ::= x \mid \top \mid \lambda x. E \mid V_1 , , V_2$		

$E \rightsquigarrow E'$	<i>(Operational semantics of Dunfield's calculus)</i>	
$\frac{\text{DSTEP-APPL} \quad E_1 \rightsquigarrow E'_1}{E_1 E_2 \rightsquigarrow E'_1 E_2}$	$\frac{\text{DSTEP-APPR} \quad E_2 \rightsquigarrow E'_2}{V_1 E_2 \rightsquigarrow V_1 E'_2}$	$\frac{\text{DSTEP-BETA}}{(\lambda x. E) V \rightsquigarrow E[x \mapsto V]}$
$\frac{\text{DSTEP-FIX}}{\mathbf{fix} x. E \rightsquigarrow E[x \mapsto \mathbf{fix} x. E]}$	$\frac{\text{DSTEP-MERGEL} \quad E_1 \rightsquigarrow E'_1}{E_1 , , E_2 \rightsquigarrow E'_1 , , E_2}$	$\frac{\text{DSTEP-MERGER} \quad E_2 \rightsquigarrow E'_2}{V_1 , , E_2 \rightsquigarrow V_1 , , E'_2}$
$\frac{\text{DSTEP-UNMERGEL}}{E_1 , , E_2 \rightsquigarrow E_1}$	$\frac{\text{DSTEP-UNMERGER}}{E_1 , , E_2 \rightsquigarrow E_2}$	$\frac{\text{DSTEP-SPLIT}}{E \rightsquigarrow E , , E}$

■ **Figure 1** The syntax and non-deterministic small-step semantics of Dunfield's calculus.

Union types are not supported by $\lambda_i^!$, since it is based on the λ_i calculus [20] with disjoint intersection types, which does not have unions either. For a better comparison, we adjust the syntax and omit union types in Dunfield's system.

Operational Semantics. The bottom part of Figure 1 presents the reduction rules. The interesting construct is the merge operator, as all other rules not involving the merge operator are standard call-by-value reduction rules. The reduction of a merge construct in Dunfield's calculus is quite flexible: a merge of two expressions (which do not even need to be two values) can step to its left subexpression (by rule DSTEP-UNMERGEL) or the right one (by rule DSTEP-UNMERGER). Any expressions can split into two by rule DSTEP-SPLIT. Therefore, even though the reduction rules may have already reached a value form, it is still possible to step further using rule DSTEP-SPLIT.

Problem 1: No Subject-Reduction. A major problem of this operational semantics is that it does not preserve types. Note that reduction is oblivious of types, so a term can reduce to two other terms with potentially different (and unrelated) types. For instance:

$$1 , , \text{True} \rightsquigarrow 1 \quad 1 , , \text{True} \rightsquigarrow \text{True}$$

Here the merge of an integer and a boolean is reduced to either the integer (using rule DSTEP-UNMERGEL) or the boolean (using rule DSTEP-UNMERGER). In Dunfield's calculus the term $1 , , \text{True}$ can have multiple types, including `Int` or `Bool` or even `Int & Bool`. As a consequence, the semantics is not type-preserving in general.

What is worse, a well-typed expression can reduce to an expression that is ill-typed:

$$(1 , , \lambda x. x + 1) 2 \rightsquigarrow 1 2$$

This reduction leads to an ill-typed term (with any type) because we drop the lambda instead of the 1 in the merge.

Problem 2: Non-determinism. Even in the case of type-preserving reductions there can be another problem. Because of the pair of unmerge rules (rule DSTEP-UNMERGEL and rule DSTEP-UNMERGER), the choice between a merge always has two options. This means that a reduced term can lead to two other terms of the same type, but with different meanings. For example:

$$1, , 2 \rightsquigarrow 1 \quad 1, , 2 \rightsquigarrow 2$$

There is even a third option to reduce a merge with the split rule (rule DSTEP-SPLIT):

$$1, , 2 \rightsquigarrow (1, , 2), , (1, , 2)$$

In other words the semantics is non-deterministic.

Note that Dunfield’s operational semantics is an overapproximation of the intended behavior. In her work, it is used to provide a soundness result for her elaboration semantics, which is type-safe (but still ambiguous).

2.3 A Type-Driven Semantics for Type Preservation

An essential problem is that the semantics cannot ignore the types if the reduction is meant to be type-preserving. Dunfield notes that “*For type preservation to hold, the operational semantics would need access to the typing derivation*” [23]. To avoid run-time type-checking, we design a type-driven semantics and use type annotations to guide reduction. Therefore our λ_i^{\dagger} calculus is explicitly typed, unlike Dunfield’s calculus. Nevertheless, it is easy to design source languages that infer some of the type annotations and insert them automatically to create valid λ_i^{\dagger} terms as we will see in Section 5. We discuss the main challenges and key ideas of the design of λ_i^{\dagger} next.

Type-driven Reduction. Our operational semantics follows a standard call-by-value small-step reduction and it is closely related to Dunfield’s semantics. However, type annotations play an important role in the reduction rules and are used to guide reduction. For example, in λ_i^{\dagger} we can write explicitly annotated expressions such as $(1, , \text{True}) : \text{Int}$ and $(1, , \text{True}) : \text{Bool}$. For those expressions the following reductions are valid:

$$(1, , \text{True}) : \text{Int} \leftrightarrow 1 \quad (1, , \text{True}) : \text{Bool} \leftrightarrow \text{True}$$

In contrast the following reductions are not possible:

$$(1, , \text{True}) : \text{Bool} \not\leftrightarrow 1 \quad (1, , \text{True}) : \text{Int} \not\leftrightarrow \text{True}$$

Note also that in λ_i^{\dagger} the meaning of expression $1, , \text{True}$ without any type annotation can only be a corresponding value $1, , \text{True}$ that does not drop any information.

Typed Reduction. The crucial component in the operational semantics that enables the use of type information during reduction is an auxiliary *typed reduction* relation $v \hookrightarrow_A v'$ that is used when we want some value to match a type. Typed reduction is where type information from annotations in λ_i^{\dagger} “filters” reductions that are invalid due to a type mismatch. Typed reduction takes a value and a type (which can be viewed as inputs), and gives a unique value of that type as output. Note that this process may result in further reduction of the value, unlike many other languages where values can never be further reduced. Typed reduction is used in two places during reduction:

$$\frac{\text{STEP-ANNOV} \quad v \hookrightarrow_A v'}{v : A \hookrightarrow v'} \qquad \frac{\text{STEP-BETA} \quad v \hookrightarrow_A v'}{(\lambda x. e : A \rightarrow B) v \hookrightarrow (e[x \mapsto v']) : B}$$

The first place where typed reduction is used is in rule STEP-ANNOV. When reduction encounters a value with a type annotation A it uses typed reduction to do further reduction depending on the type A . To see typed reduction in action, consider a simple merge of primitive values such as 1 , True , c with an annotation $\text{Int} \& \text{Char}$. Using rule STEP-ANNOV typed reduction is invoked, resulting in:

$$1, \text{True}, c \hookrightarrow_{\text{Int} \& \text{Char}} 1, c$$

We could have type-reduced the same value under a similar type but where the two types in the intersection are interchanged:

$$1, \text{True}, c \hookrightarrow_{\text{Char} \& \text{Int}} c, 1$$

Both typed reductions are valid and they illustrate the ability of typed reduction to create a value that matches exactly with the shape of the type.

The second place where typed reduction is used is in rule STEP-BETA. In a function application, the actual argument could be a merge containing more components than what the function expects. One example is $(\lambda x. x + 1 : \text{Int} \rightarrow \text{Int}) (1, \text{True})$. Since the merge term $(1, \text{True})$ provides an integer 1 , the redundant components (the True in this case) are useless, and sometimes even harmful. Consider a function $\lambda x. (x, \text{False})$ with type $\text{Int} \rightarrow \text{Int} \& \text{Bool}$, applied to $(1, \text{True})$. If we performed direct substitution of the argument in the lambda body, this would result in $1, \text{True}, \text{False}$. This brings ambiguity, and the term is not well-typed, as we shall see in Section 2.5. Therefore, before substitution, the value must be further reduced with typed reduction under the expected type of the function argument. Thus the value that is substituted in the lambda body is 1 (but not $1, \text{True}$), and the final result is $1, \text{False}$.

These examples show some non-trivial aspects of typed reduction, which must decompose values, and possibly drop some of the components and permute other components. The details of the typed reduction relation will be discussed in Section 4. As we shall see functions introduce further complications.

2.4 The Challenges of Functions

One of the hardest challenges in designing the semantics of λ_i^{\dagger} was the design of the rules for functions. We discuss the challenges next.

Return Types Matter. As we have seen above, the input type annotation of lambdas is necessary during beta reduction. However, it is not enough to distinguish among multiple functions in a merge (e.g. $(\lambda x. x + 1)$, $(\lambda x. \text{True})$) without run-time type checking. Unlike primitive values, whose types can be told by their forms, for functions, we need the type of the function (including the output type) to select the right function from a merge. Therefore, in λ_i^{\dagger} all functions are annotated with both the input and output types. With such annotations we can deal with programs such as:

$$((\lambda f. f \ 1) : (\text{Int} \rightarrow \text{Int}) \rightarrow \text{Int}) ((\lambda x. x + 1) : \text{Int} \rightarrow \text{Int}, (\lambda x. \text{True}) : \text{Int} \rightarrow \text{Bool})$$

In this program we have a lambda that takes a function f as an argument and applies it to 1. The lambda is applied to the merge of two functions of types $\text{Int} \rightarrow \text{Int}$ and $\text{Int} \rightarrow \text{Bool}$. To select the right function from the merge, the types of the functions are used to guide the reduction of the merge. This avoids the need for run-time type-checking, which would be otherwise necessary to recover the full type of functions.

Annotation Refinement. Given a value, for any of its supertypes, typed reduction gives a result. Since functions are values, sometimes this leads to the refinement of the type annotation of lambdas. Following the convention introduced by previous works [20], \rightarrow has lower precedence than $\&$, which means $A \rightarrow B \& C$ equals to $A \rightarrow (B \& C)$. Consider a single function $\lambda x. x, , \text{True} : \text{Int} \rightarrow \text{Int} \& \text{Bool}$ to be reduced under type $\text{Int} \& \text{Bool} \rightarrow \text{Int}$. To let the function return an integer when applied to a merge of type $\text{Int} \& \text{Bool}$, we must change either the lambda body or the embedded annotation. Since reducing under a lambda body is not allowed in call-by-value, λ_i^{\dagger} adopts the latter option, and treats the input and output annotations differently. The input annotation should not be changed as it represents the expected input type of the function and helps to adjust the input value before substitution in beta reduction. The output annotation, in contrast, must be replaced by Int , representing a future reduction to be done after substitution. The output of the application then can be thought as an integer and can be safely merged with another boolean, for example. In short, the actual λ_i^{\dagger} reduction is:

$$\begin{aligned} & ((\lambda x. x, , \text{True}) : \text{Int} \rightarrow \text{Int} \& \text{Bool}) : \text{Int} \& \text{Bool} \rightarrow \text{Int} \\ & \hookrightarrow (\lambda x. x, , \text{True}) : \text{Int} \rightarrow \text{Int} \end{aligned}$$

2.5 Disjoint Intersection Types and Consistency for Determinism

Even if the semantics is type-directed and it rules out reductions that do not preserve types, it can still be non-deterministic. To solve this problem, we employ the disjointness restriction proposed by Oliveira et al. [20] and the novel notion of *consistency*. Both disjointness and consistency play a fundamental role in the proof of determinism.

Disjointness. Two types are disjoint (written as $A * B$), if any common supertypes that they have are *top-like types* (i.e. supertypes of any type; written as $\lceil C \rceil$).

► **Definition 1** (Disjoint Types).

$$A * B \equiv \forall C, \text{ if } A <: C \text{ and } B <: C \text{ then } \lceil C \rceil$$

Intuitively, if two types are disjoint (e.g. $\text{Int} \& \text{Char} * \text{Bool}$), their corresponding values do not overlap (e.g. 1, , 'c' and True). The only exceptions are top-like types, as they are disjoint with any types [2]. Since every value of a top-like type has the same effect, typed reduction unifies them to a fixed result. Thus the disjointness checking in the following typing rule guarantees that e_1 and e_2 can be merged safely, without any ambiguities. For example, this typing rule does not accept 1, , 2 or True, , 1, , False, as two subterms of the merge have overlapped types (in this case, the same type Int and Bool , respectively).

$$\frac{\Gamma \vdash e_1 : A \quad \Gamma \vdash e_2 : B \quad A * B}{\Gamma \vdash e_1, , e_2 : A \& B} \text{ETYP-MERGE}$$

Consistency. Recall the split rule (rule DSTEP-SPLIT) in Dunfield’s semantics: $E \rightsquigarrow E, E$. It duplicates terms in a merge. Similar things can happen in our typed reduction if the type has overlapping parts, which is allowed, for example, in an expression $1 : \text{Int} \& \text{Int}$. Note that in this expression the term 1 can be given type annotation $\text{Int} \& \text{Int}$ since $\text{Int} <: \text{Int} \& \text{Int}$. During reduction, typed reduction is eventually used to create a value that matches the shape of type $\text{Int} \& \text{Int}$ by duplicating the integer:

$$1 \hookrightarrow_{\text{Int} \& \text{Int}} 1, 1$$

Note that the disjointness restriction does not allow sub-expressions in a merge to have the same type: $1, 1$ cannot type-check with rule ETYP-MERGE. To obtain *type preservation*, there is a special (run-time) typing rule for merges of values, where a novel consistency check is used (written as $v_1 \approx v_2$):

$$\frac{\cdot \vdash v_1 : A \quad \cdot \vdash v_2 : B \quad v_1 \approx v_2}{\Gamma \vdash v_1, v_2 : A \& B} \text{ETYP-MERGEV}$$

Mainly, consistency allows values to have overlapped parts as far as they are syntactically equal. For example, $1, \text{True}$ and $1, 'c'$ are consistent, since the overlapped part Int in both of merges is the same value. True and $'c'$ are consistent because they are not overlapped at all. But $1, \text{True}$ and 2 are *not consistent*, as they have different values for the same type Int . When two values have disjoint types, they must be consistent. For merges of such values, both rule ETYP-MERGEV and rule ETYP-MERGE can be applied, and the types always coincide. In λ_i^{\dagger} , consistency is defined in terms of typed reduction:

► **Definition 2 (Consistency).** *Two values v_1 and v_2 are said to be consistent (written $v_1 \approx v_2$) if, for any type A , the result of typed reduction for the two values is the same.*

$$v_1 \approx v_2 \equiv \forall A, \text{ if } v_1 \hookrightarrow_A v'_1 \text{ and } v_2 \hookrightarrow_A v'_2 \text{ then } v'_1 = v'_2$$

Although the specification of consistency is decidable and an equivalent algorithmic definition exists, it is not required. In practice, in a programming language implementation, the rule ETYP-MERGEV may be omitted, since, as stated, its main purpose is to ensure that run-time values are type-preserving.

Finally, note that λ_i [20] is stricter than λ_i^{\dagger} and forbids any intersection types which are not disjoint. That is to say, the term $1 : \text{Int} \& \text{Int}$ is not well-typed because the intersection $\text{Int} \& \text{Int}$ is not disjoint. The idea of allowing unrestricted intersections, while only having the disjointness restriction for merges, was first employed in the NeColus calculus [5]. λ_i^{\dagger} follows such an idea and $1 : \text{Int} \& \text{Int}$ is well-typed in λ_i^{\dagger} . Allowing unrestricted intersections adds extra expressive power. For instance, in calculi with polymorphism, unrestricted intersections can be used to encode *bounded quantification* [8], whereas with disjoint intersections only such an encoding does not work [6].

3 The λ_i^{\dagger} Calculus: Syntax, Subtyping and Typing

This section presents the syntax, subtyping, and typing of λ_i^{\dagger} : a calculus with intersection types and a merge operator. This calculus is a small variant of the λ_i calculus [20] (which itself is inspired by Dunfield’s calculus [23]) extended with *annotated expressions*, *explicit subsumption* and *fixpoints*. The explicit type annotations and subtyping are necessary for the type-directed operational semantics of λ_i^{\dagger} and to preserve determinism. The addition of fixpoints illustrates the ability of TDOS to deal with non-terminating programs, which are still not supported by calculi that rely on elaboration and semantic coherence proofs [5, 6].

3.1 Syntax

The syntax of λ_i is:

<i>Type</i>	$A, B ::= \text{Int} \mid \text{Top} \mid A \rightarrow B \mid A \& B$
<i>Expr</i>	$e ::= x \mid i \mid \top \mid e : A \mid e_1 e_2 \mid \lambda x. e : A \rightarrow B \mid e_1 , , e_2 \mid \mathbf{fix} \ x. e : A$
<i>Value</i>	$v ::= i \mid \top \mid \lambda x. e : A \rightarrow B \mid v_1 , , v_2$
<i>Context</i>	$\Gamma ::= \cdot \mid \Gamma, x : A$

Types. Meta-variables A and B range over types. Two basic types are included: the integer type Int and the top type Top . Function types $A \rightarrow B$ and intersection types $A \& B$ can be used to construct compound types.

Expressions. Meta-variable e ranges over expressions. Expressions include some standard constructs: variables (x); integers (i); a canonical top value \top ; annotated expressions ($e : A$); and application of a term e_1 to term e_2 (denoted by $e_1 e_2$). Lambda abstractions ($\lambda x. e : A \rightarrow B$) must have a type annotation $A \rightarrow B$, meaning that the input type is A and the output type is B . The expression $e_1 , , e_2$ is the merge of expressions e_1 and e_2 . Finally, fixpoints $\mathbf{fix} \ x. e : A$ (which also require a type annotation) model recursion.

Values and Contexts. The meta-variable v ranges over values. Values include integers, the canonical \top value, lambda abstractions and merges of values. Typing contexts are standard. Γ tracks the bound variables x with their type A .

3.2 Subtyping and Disjointness

The subtyping rules of the form $A <: B$ are shown on the top of Figure 2. These subtyping rules, except for rule S-TOPARR, were first introduced by Davies and Pfenning [21], and are used in λ_i as well. The original subtyping relation is known to be reflexive and transitive [21]. We proved the reflexivity and transitivity of the extended subtyping relation as well. There are 3 rules regarding intersection types. Together they define $A \& B$ as the greatest lower bound of A and B .

Top-like Types and Arrow Types. Intuitively, a top-like type is both a supertype and a subtype of Top , including the Top type and intersections of top-like types. The newly added rule S-TOPARR enlarges top-like types to include arrow types when their return types are top-like. A simple unary relation that captures top-like types inductively is defined on the bottom of Figure 2. The following theorem states the correctness and completeness of the definition.

► **Lemma 3** (Soundness and Completeness of the Definition of Top-like Types).

$$\lceil A \rceil \text{ if and only if } \text{Top} <: A$$

Rule S-TOPARR is inspired by the following rule in BCD-style subtyping [3] (and adopted by Bi et al. [5]):

$$\frac{}{\text{Top} <: \text{Top} \rightarrow \text{Top}} \text{BCD-TOPARR}$$

26:12 A Type-Directed Operational Semantics for a Calculus with a Merge Operator

$$\boxed{A <: B} \quad (Subtyping)$$

$$\begin{array}{c}
 \text{S-z} \\
 \hline
 \text{Int} <: \text{Int}
 \end{array}
 \quad
 \begin{array}{c}
 \text{S-TOP} \\
 \hline
 A <: \text{Top}
 \end{array}
 \quad
 \begin{array}{c}
 \text{S-ARR} \\
 \frac{B_1 <: A_1 \quad A_2 <: B_2}{A_1 \rightarrow A_2 <: B_1 \rightarrow B_2}
 \end{array}
 \quad
 \begin{array}{c}
 \text{S-ANDR} \\
 \frac{A_1 <: A_2 \quad A_1 <: A_3}{A_1 <: A_2 \& A_3}
 \end{array}$$

$$\begin{array}{c}
 \text{S-TOPARR} \\
 \frac{\text{Top} <: B_2}{A <: B_1 \rightarrow B_2}
 \end{array}
 \quad
 \begin{array}{c}
 \text{S-ANDL1} \\
 \frac{A_1 <: A_3}{A_1 \& A_2 <: A_3}
 \end{array}
 \quad
 \begin{array}{c}
 \text{S-ANDL2} \\
 \frac{A_2 <: A_3}{A_1 \& A_2 <: A_3}
 \end{array}$$

$$\boxed{\lceil A \rceil} \quad (Top-like types)$$

$$\begin{array}{c}
 \text{TL-AND} \\
 \frac{\lceil A \rceil \quad \lceil B \rceil}{\lceil A \& B \rceil}
 \end{array}
 \quad
 \begin{array}{c}
 \text{TL-TOP} \\
 \frac{}{\lceil \text{Top} \rceil}
 \end{array}
 \quad
 \begin{array}{c}
 \text{TL-ARR} \\
 \frac{\lceil B \rceil}{\lceil A \rightarrow B \rceil}
 \end{array}$$

■ **Figure 2** Subtyping rules of λ_i and definition of top-like types.

Since BCD-style subtyping includes a transitivity rule as an axiom, with this rule, $\text{Int} \rightarrow \text{Top}$ and $\text{Int} \rightarrow (\text{Top} \rightarrow \text{Top})$ are supertypes (and also subtypes) of Top . Due to the lack of built-in transitivity rule in λ_i 's subtyping, the above consequence has to be expressed more explicitly in the adapted rule S-TOPARR. We will come back to our motivation for including rule S-TOPARR in Section 3.3.

Disjointness. In Section 2.5, the specification of disjointness is presented. Such specification is a slightly more liberal version of the definition originally used in λ_i . In particular in our definition A and B themselves can be *top-like types*, which was forbidden in λ_i . An equivalent algorithmic definition of disjointness ($A *_a B$) is presented in Appendix A, which is the same as the definition in the NeColus calculus [5].

► **Lemma 4** (Disjointness Properties). *Disjointness satisfies:*

1. $A * B$ if and only if $A *_a B$.
2. if $A * (B_1 \rightarrow C)$ then $A * (B_2 \rightarrow C)$.
3. if $A * B \& C$ then $A * B$ and $A * C$.

3.3 Typing

The expression typing judgment $\Gamma \vdash e : A$ is standard. It says that in the typing environment Γ the expression e is of type A . Unlike λ_i , there is no well-formedness restriction on types¹. This generalization is inspired by the calculus NeColus [5], where the well-formedness constraints are removed from λ_i , and expressions like $1 : \text{Int} \& \text{Int}$ are allowed. In other words the calculus supports *unrestricted intersections* as well as *disjoint intersection types* (which are the only kind of intersections supported in λ_i).

The type system, shown in Figure 3, is syntax-directed. Most typing rules directly follow the declarative type system of λ_i , including the merge rule ETYP-MERGE, where disjointness is used. When two expressions have disjoint types, any parts from each of them do not have overlapping types. Therefore, their merge does not introduce ambiguity. With this

¹ The wellformedness and typing rules for λ_i can be found in Section 5.2.

$\Gamma \vdash e : A$				(Typing)
$\frac{}{\Gamma \vdash \top : \text{Top}}$	$\frac{}{\Gamma \vdash i : \text{Int}}$	$\frac{x : A \in \Gamma}{\Gamma \vdash x : A}$	$\frac{\Gamma \vdash e : B \quad B <: A}{\Gamma \vdash (e : A) : A}$	
$\frac{\Gamma, x : A \vdash e : B \quad C <: A \quad B <: D}{\Gamma \vdash (\lambda x. e : A \rightarrow D) : C \rightarrow D}$		$\frac{}{\Gamma \vdash (\mathbf{fix} x. e : A) : A}$	$\frac{\Gamma \vdash e_1 : A \rightarrow B \quad \Gamma \vdash e_2 : A}{\Gamma \vdash e_1 e_2 : B}$	
$\frac{\Gamma \vdash e_1 : A \quad \Gamma \vdash e_2 : B \quad A * B}{\Gamma \vdash e_1 , , e_2 : A \& B}$		$\frac{\cdot \vdash v_1 : A \quad \cdot \vdash v_2 : B \quad v_1 \approx v_2}{\Gamma \vdash v_1 , , v_2 : A \& B}$		

■ **Figure 3** Type system of λ_i^{\dagger} .

restriction, rule ETYP-MERGE does not accept expressions like $1, , 2$ or even $1, , 1$. On the other hand, the novel rule ETYP-MERGEV allows *consistent* values to be merged regardless of their types. It accepts $1, , 1$ while still rejecting $1, , 2$. It is for values only, and values are closed. Therefore the type judgments appearing in it as premises should have empty context, which is denoted by \cdot . Together the two rules support the determinism and type preservation of the TDOS, as discussed in Section 2.5.

Top-Like Types and Merges of Functions. We can finally come back to the motivation to include rule S-TOPARR in subtyping and depart from both Dunfield calculus and λ_i , which do not have such a rule. *Without the rule S-TOPARR in subtyping*, no arrow types are top-like, therefore two arrow types $A \rightarrow B$ and $C \rightarrow D$ are never disjoint in terms of Definition 1, as they have a common supertype $A \& C \rightarrow \text{Top}$. Consequently, we can never create merges with more than one function, which is quite restrictive. For Dunfield this is not a problem, because she does not have the disjointness restriction. So her calculus supports merges of any functions (but it is incoherent). In λ_i an ad-hoc solution is proposed, by forcing the matter and employing the syntactic definition of top-like types in Figure 2 in disjointness. However this means that in λ_i Lemma 3 is false, since top-like function types are not supertypes of Top . In contrast, the approach we take in λ_i^{\dagger} is to add the rule S-TOPARR in subtyping. Now $\text{Top} <: (A \& C \rightarrow \text{Top})$ is derivable and thus $A \& C \rightarrow \text{Top}$ is genuinely a top-like type. In turn this makes merges of multiple functions typeable without losing the intuition behind top-like types.

Type-Checking for Lambda Abstractions. Rule ETYP-ABS can be thought as a combination of the standard typing rule and the subsumption rule. A well-typed lambda abstraction can have multiple types with the same return type. Its type annotation indicates the *principal input type* and the return type. Thus the input type can be any subtype of the principal one, since arrow types are contravariant in their argument types. While the principal input type describes the lambda’s expectation on its argument, the annotated return type ensures the type of the evaluated result of lambdas. It just needs to be a supertype of the inner expression of the lambda. Rule ETYP-ABS is inspired by the “distributed” use of subsumption in the $\lambda\&$ calculus [9].

Explicit Subsumption. Unlike many calculi where there is a general subsumption rule that can apply anywhere, in λ_i^{\dagger} subsumption needs to be explicitly triggered by a type annotation (except for lambdas, as explained above). The annotation rule ETYP-ANNO acts as explicit subsumption and assigns supertypes to expressions, provided a suitable type annotation. There is a strong motivation not to include a general (implicit) subsumption rule in calculi with disjoint intersection types. With an implicit subsumption rule disjointness alone is insufficient to prevent some ambiguous terms, as shown in the following example.

$$\begin{array}{c}
 \text{SUBSUMPTION} \frac{\cdot \vdash 1 : \text{Int} \quad \text{Int} <: \text{Top}}{\cdot \vdash 1 : \text{Top}} \quad \cdot \vdash 2 : \text{Int} \quad \text{Top} * \text{Int} \\
 \text{ETYP-MERGE} \frac{\cdot \vdash 1 : \text{Top} \quad \cdot \vdash 2 : \text{Int} \quad \text{Top} * \text{Int}}{\cdot \vdash 1, , 2 : \text{Top} \& \text{Int}} \\
 \text{SUBSUMPTION} \frac{\cdot \vdash 1, , 2 : \text{Top} \& \text{Int} \quad \text{Top} \& \text{Int} <: \text{Int} \& \text{Int}}{\cdot \vdash 1, , 2 : \text{Int} \& \text{Int}}
 \end{array}$$

Via the typical implicit subsumption, type `Top` is assigned to integer 1. Then 1 can be merged with 2 of type `Int` since their types are disjoint. At that time, the merged term `1, , 2` has type `Top & Int`, which is a subtype of `Int & Int`. By applying the subsumption rule again, the ambiguous term `1, , 2` finally bypasses the disjointness restriction, having type `Int & Int`. However, note that with rule ETYP-ANNO we can still type-check the term $(1 : \text{Top}), , 2$, and reducing that term under the type `Int` can only unambiguously result in 2. The type annotation is key to prevent using the value 1 as an integer. Finally, the use of an explicit subsumption rule is a simpler alternative to bidirectional type-systems employed in other calculi with disjoint intersection types. Bidirectional type-checking is also capable of controlling subsumption, but adds more complexity.

Principal Types. The principal type of a value is the most specific one among all of its types, i.e. it is the subtype of any other type of the term. Its definition is syntax-directed.

► **Definition 5** (Principal types). $type_p\langle v \rangle$ calculates the principal type of value v .

$$\begin{aligned}
 type_p\langle \top \rangle &= \text{Top} \\
 type_p\langle n \rangle &= \text{Int} \\
 type_p\langle \lambda x. e : A \rightarrow B \rangle &= A \rightarrow B \\
 type_p\langle v_1, , v_2 \rangle &= type_p\langle v_1 \rangle \& type_p\langle v_2 \rangle
 \end{aligned}$$

► **Lemma 6** (Principal Types). For any value v , if its principal type is A , then

1. if $\cdot \vdash v : B$ then $A <: B$.
2. if $\cdot \vdash v : B$ and $B * C$ then $A * C$.
3. $\cdot \vdash v : A$.

4 A Type-Directed Operational Semantics for λ_i^{\dagger}

This section introduces the type-directed operational semantics for λ_i^{\dagger} . The operational semantics uses type information arising from type annotations to guide the reduction process. In particular, a new relation called *typed reduction* is used to further reduce values based on the contextual type information, forcing the value to match the type structure. We show two important properties for λ_i^{\dagger} : *determinism of reduction* and *type soundness*. That is to say, there is only one way to reduce an expression according to the small-step relation, and the process preserves types and never gets stuck.

$$\boxed{v \hookrightarrow_A v'} \quad (\textit{Typed reduction})$$

$$\begin{array}{c}
\text{TREDUCE-LIT} \\
\hline
i \hookrightarrow_{\textit{Int}} i
\end{array}
\quad
\begin{array}{c}
\text{TREDUCE-TOP} \\
\hline
v \hookrightarrow_{\textit{Top}} \top
\end{array}
\quad
\begin{array}{c}
\text{TREDUCE-TOPARR} \\
\hline
\top \mid B \mid \\
v \hookrightarrow_{A \rightarrow B} \lambda x. \top : \textit{Top} \rightarrow B
\end{array}$$

$$\begin{array}{c}
\text{TREDUCE-ARROW} \\
\hline
\frac{\neg \mid D \mid \quad C <: A \quad B <: D}{\lambda x. e : A \rightarrow B \hookrightarrow_{C \rightarrow D} \lambda x. e : A \rightarrow D}
\end{array}
\quad
\begin{array}{c}
\text{TREDUCE-AND} \\
\hline
\frac{v \hookrightarrow_A v_1 \quad v \hookrightarrow_B v_2}{v \hookrightarrow_{A \& B} v_1, v_2}
\end{array}$$

$$\begin{array}{c}
\text{TREDUCE-MERGEVL} \\
\hline
\frac{v_1 \hookrightarrow_A v'_1 \quad A \textit{ ordinary}}{v_1, v_2 \hookrightarrow_A v'_1}
\end{array}
\quad
\begin{array}{c}
\text{TREDUCE-MERGEVR} \\
\hline
\frac{v_2 \hookrightarrow_A v'_2 \quad A \textit{ ordinary}}{v_1, v_2 \hookrightarrow_A v'_2}
\end{array}$$

■ **Figure 4** Typed reduction of λ_i^i .

4.1 Typed Reduction of Values

To account for the type information during reduction λ_i^i uses an auxiliary reduction relation called *typed reduction* for reducing values under a certain type. Typed reduction $v \hookrightarrow_A v'$ reduces the value v under type A , producing a value v' that has type A . It arises when given a value v of some type, where A is a supertype of the type of v , and v needs to be converted to a value compatible with the supertype A . The typed reduction ensures that values and types have a strong correspondence. If a value is well-typed, its principal type can be told directly by looking at its syntactic form.

Figure 4 shows the typed reduction relation. Rule TREDUCE-TOP expresses the fact that \textit{Top} is the supertype of any type, which means that any value can be reduced under type \textit{Top} . Similarly, rule TREDUCE-TOPARR indicates that any value reduces to a lambda abstraction $\lambda x. \top : \textit{Top} \rightarrow B$ under a top-like arrow type $A \rightarrow B$. Although it is not the only inhabited value of type $A \rightarrow B$, the reduction result has to be fixed for determinism. Rule TREDUCE-LIT expresses that an integer value reduced under the supertype \textit{Int} is just the integer value itself. Rule TREDUCE-ARROW states that a lambda value $\lambda x. e : A \rightarrow B$, under a *non-top-like type* $C \rightarrow D$, evaluates to $\lambda x. e : A \rightarrow D$ if $C <: A$ and $B <: D$. The restriction that $C \rightarrow D$ is not top-like avoids overlapping with rule TREDUCE-TOPARR. Importantly rule TREDUCE-ARROW changes the return type of lambda abstractions. For example:

$$(\lambda x. x, , 2 : \textit{Char} \rightarrow \textit{Char} \& \textit{Int}) \hookrightarrow_{(\textit{Char} \& \textit{Int} \rightarrow \textit{Char})} \lambda x. x, , 2 : \textit{Char} \rightarrow \textit{Char}$$

Intersections and Merges. In the remaining rules, we first decompose intersections. Then we only need to consider types that are not intersections, which are called *ordinary types* [21]:

$$\boxed{A \textit{ ordinary}} \quad (\textit{Ordinary types})$$

$$\begin{array}{c}
\text{O-TOP} \\
\hline
\textit{Top ordinary}
\end{array}
\quad
\begin{array}{c}
\text{O-INT} \\
\hline
\textit{Int ordinary}
\end{array}
\quad
\begin{array}{c}
\text{O-ARROW} \\
\hline
A \rightarrow B \textit{ ordinary}
\end{array}$$

We take care of the value by going through every merge, until both value and types are in a basic form. Rule `TREDUCE-MERGEVL` and rule `TREDUCE-MERGEVR` are a pair of rules for reducing merges under an ordinary type. Since the type is not an intersection, the result contains no merge. Usually, we need to select between the left part and right part of a merge according to the type. The values of disjoint types do not overlap on non-top-like types. For example, $1, (\lambda x. x : \text{Int} \rightarrow \text{Int}) \hookrightarrow_{\text{Int}} 1$ selects the left part. For top-like types, no matter which rule is applied, the reduction result is determined by the type only, as rule `TREDUCE-TOP` and rule `TREDUCE-TOPARR` suggest.

Rule `TREDUCE-AND` is the rule that deals with intersection types. It says that if a value v can be reduced to v_1 under type A , and can be reduced to v_2 under type B , then its reduction result under type $A \& B$ is the merge of two results v_1, v_2 . Note that this rule may *duplicate values*. For example $1 \hookrightarrow_{\text{Int} \& \text{Int}} 1, 1$. Such duplication requires special care, since the merge violates disjointness. The specially designed typing rule (rule `ETYP-MERGEV`) uses the notion of consistency (discussed in Section 4.2) instead of disjointness to type-check a merge of two values. Note also that such duplication implies that sometimes it is possible to use either rule `TREDUCE-MERGEVL` or rule `TREDUCE-MERGEVR` to reduce a value. For example, $1, 1 \hookrightarrow_{\text{Int}} 1$. The consistency restriction (Definition 2) in rule `ETYP-MERGEV` ensures that no matter which rule is applied in such a case, the result is the same.

Example. A larger example to demonstrate how typed reduction works is:

$$\begin{aligned} & (\lambda x. x, , 'c' : \text{Int} \rightarrow \text{Int} \& \text{Char}), , (\lambda x. x : \text{Bool} \rightarrow \text{Bool}), , 1 \\ & \hookrightarrow_{\text{Int} \& (\text{Int} \rightarrow \text{Int})} 1, , (\lambda x. x, , 'c' : \text{Int} \rightarrow \text{Int}) \end{aligned}$$

The initial value is the merge of two lambda abstractions and an integer. The target type is $\text{Int} \& (\text{Int} \rightarrow \text{Int})$. Because the target type is an intersection, typed reduction first employs rule `TREDUCE-AND` to decompose the intersection into Int and $\text{Int} \rightarrow \text{Int}$. Under type Int the value reduces to 1 , and under type $\text{Int} \rightarrow \text{Int}$ it will reduce to $\lambda x. x, , 'c' : \text{Int} \rightarrow \text{Int}$. Therefore, we obtain the merge $1, , (\lambda x. x, , 'c' : \text{Int} \rightarrow \text{Int})$ with type $\text{Int} \& (\text{Int} \rightarrow \text{Int})$.

Basic Properties of Typed Reduction. Some properties of typed reduction can be proved directly by induction on the typed reduction derivation. First, when typed reduction is under a top-like type, the result only depends on the type. Second, typed reduction produces the same result whenever it is done directly or indirectly. Third, if a well-typed value can be typed reduced by some type, its principal type must be a subtype of that type.

► **Lemma 7** (Typed reduction on top-like types). *If $\lambda A[, v_1 \hookrightarrow_A v'_1$, and $v_2 \hookrightarrow_A v'_2$ then $v'_1 = v'_2$.*

When typed reduction is under a top-like type, the result only depends on the type.

► **Lemma 8** (Transitivity of typed reduction). *If $v \hookrightarrow_A v_1$, and $v_1 \hookrightarrow_B v_2$, then $v \hookrightarrow_B v_2$.*

Typed reduction produces the same result whenever it is done directly or indirectly.

► **Lemma 9** (Typed reduction respects subtyping). *If $v \hookrightarrow_A v'$, then $\text{type}_p(v) <: A$.*

This lemma relates typed reduction and subtyping. It states that if a well-typed value can be typed reduced by type A , its principal type must be a subtype of A .

4.2 Consistency and Type Soundness of Typed Reduction

Consistent values, as specified in Definition 2, introduce no ambiguity in typed reduction. Consider one type, if two consistent values both can reduce under the type, they should produce the same result. The *consistency* restriction ensures that duplicated values in a merge type-check, but it still rejects merges with different values of the same type. A value of a top-like type is consistent with any other value. It only type reduces under top-like types, which leads to a fixed result decided by the type.

Relating Disjointness and Consistency. Assuming that two values have disjoint types, according to Lemma 6, their principal types must be disjoint as well. From Lemma 9, we can conclude that when the two values both reduce under a type, that type must be a common supertype of their principal types, which is known to be top-like (Definition 1). Furthermore, Lemma 7 implies that their reduction results are always the same under such top-like types, so they are consistent (Definition 2).

► **Lemma 10** (Consistency of disjoint values). *If $A * B$, $\cdot \vdash v_1 : A$, and $\cdot \vdash v_2 : B$ then $v_1 \approx v_2$.*

Determinism and Type Soundness of Typed Reduction. The merge construct makes it hard to design a deterministic operational semantics. Disjointness and consistency restrictions prevent merges like 1, 2, and bring the possibility to deal with merges based on types. Typed reduction takes a well-typed value, which, if it is a merge, must be consistent (according to Lemma 10). When the two typed reduction rules for merges (rule TREDUCE-MERGEVL and rule TREDUCE-MERGEVR) overlap, no matter which one is chosen, either value reduces to the same result due to consistency (Definition 2). Indeed our typed reduction relation always produces a unique result for any legal combination of the input value and type. This serves as a foundation for the determinism of the operational semantics.

► **Lemma 11** (Determinism of Typed Reduction). *For every well-typed v (that is there is some type B such that $\cdot \vdash v : B$), if $v \hookrightarrow_A v_1$ and $v \hookrightarrow_A v_2$ then $v_1 = v_2$.*

Via the transitivity lemma (Lemma 8) and the above determinism lemma, we obtain the following property: any reduction results of the given value are consistent.

► **Lemma 12** (Consistency after Typed Reduction). *If v is well-typed, and $v \hookrightarrow_A v_1$, and $v \hookrightarrow_B v_2$ then $v_1 \approx v_2$.*

The lemma shows that the reduction result of rule TREDUCE-AND is always made of consistent values, which is needed in type preservation via the typing rule ETYP-MERGEV. Then a (generalized) type preservation lemma on typed reduction can be proved.

► **Lemma 13** (Preservation of Typed reduction). *If $\cdot \vdash v : B$ and $v \hookrightarrow_A v'$ then $\cdot \vdash v' : A$.*

In the particular case where $A = B$, this lemma shows that typed reduction preserves types. However, more generally, it shows that if a value is well-typed under a type B and it can be type reduced under another type A then the reduced value is always well-typed at type A . Finally, the typed reduction progress lemma is:

► **Lemma 14** (Progress of Typed Reduction). *If $\cdot \vdash v : A$, and $A <: B$, then $\exists v', v \hookrightarrow_B v'$.*

$$\boxed{e \hookrightarrow e'} \quad (\text{Reduction})$$

$$\begin{array}{c}
 \text{STEP-APPL} \\
 \frac{e_1 \hookrightarrow e'_1}{e_1 e_2 \hookrightarrow e'_1 e_2} \\
 \\
 \text{STEP-MERGER} \\
 \frac{e_2 \hookrightarrow e'_2}{v_1 e_2 \hookrightarrow v_1 e'_2} \\
 \\
 \text{STEP-FIX} \\
 \frac{}{\mathbf{fix} \ x. e : A \hookrightarrow e[x \mapsto \mathbf{fix} \ x. e : A]} \\
 \\
 \text{STEP-MERSEL} \\
 \frac{e_1 \hookrightarrow e'_1}{e_1 ,, e_2 \hookrightarrow e'_1 ,, e_2} \\
 \\
 \text{STEP-MERGER} \\
 \frac{e_2 \hookrightarrow e'_2}{v_1 ,, e_2 \hookrightarrow v_1 ,, e'_2} \\
 \\
 \text{STEP-ANNO} \\
 \frac{e \hookrightarrow e'}{e : A \hookrightarrow e' : A} \\
 \\
 \text{STEP-BETA} \\
 \frac{v \hookrightarrow_A v'}{(\lambda x. e : A \rightarrow B) v \hookrightarrow (e[x \mapsto v']) : B} \\
 \\
 \text{STEP-ANNOV} \\
 \frac{v \hookrightarrow_A v'}{v : A \hookrightarrow v'}
 \end{array}$$

■ **Figure 5** Call-by-value reduction of λ_i^{\dagger} .

4.3 Reduction

The reduction rules are presented in Figure 5. Most of them are standard. Rule STEP-BETA and rule STEP-ANNOV are the two rules relying on typed reduction judgments. Rule STEP-BETA says that a lambda value $\lambda x. e : A \rightarrow B$ applied to value v reduces by replacing the bound variable x in e by v' . Importantly v' is obtained by type reducing v under type A . In other words, in rule STEP-BETA further (typed) reduction may be necessary on the argument depending on its type. This is unlike many other calculi where values are in a final form and no further reduction is needed (thus the value v can be directly substituted). The rule STEP-ANNOV says that an annotated $v : A$ can be reduced to v' if v type reduces to v' under type A .

Metatheory of Reduction. When designing the operational semantics of λ_i^{\dagger} , we want it to have two properties: *determinism of reduction* and *type soundness*. That is to say, there is only one way to reduce an expression according to the small-step relation, and the process preserves types and never gets stuck. Similar lemmas on typed reduction were already presented, which are necessary for proving the following theorems, mainly in cases related to rule STEP-ANNOV and rule STEP-BETA.

- **Theorem 15** (Determinism of \hookrightarrow). *If $\cdot \vdash e : A$, $e \hookrightarrow e_1$, $e \hookrightarrow e_2$, then $e_1 = e_2$.*
- **Theorem 16** (Type Preservation of \hookrightarrow). *If $\cdot \vdash e : A$, and $e \hookrightarrow e'$ then $\cdot \vdash e' : A$.*
- **Theorem 17** (Progress of \hookrightarrow). *If $\cdot \vdash e : A$, then e is a value or $\exists e', e \hookrightarrow e'$.*

5 Relationship to Dunfield's Calculus and λ_i

Dunfield's calculus [23] and λ_i [20] are two calculi that directly inspired λ_i^{\dagger} . In this section, we discuss the relationship between λ_i^{\dagger} and them. First, we show that λ_i^{\dagger} 's TDOS and a slightly extended version of Dunfield's non-deterministic operational semantics are related. The need for extending Dunfield's original semantics is mostly due to the addition of the rule S-TOPARR in subtyping, which Dunfield does not have. In Section 6 we also discuss a variant of λ_i^{\dagger} (which does not include rule S-TOPARR) and show that such variant requires

$$\begin{aligned}
|i| &= i \\
|\top| &= \top \\
|\lambda x. e : A \rightarrow B| &= \lambda x. |e| \\
|\mathbf{fix} x. e : A| &= \mathbf{fix} x. |e| \\
|e : A| &= |e| \\
|e_1 e_2| &= |e_1| |e_2| \\
|e_1 , , e_2| &= |e_1| , , |e_2|
\end{aligned}$$

■ **Figure 6** Type erasure for λ_i^{\dagger} expressions.

no changes to Dunfield’s original semantics. The other relationship is between λ_i^{\dagger} ’s type system and λ_i ’s type system. The former comparison shows the soundness of the operational semantics of λ_i^{\dagger} with respect to Dunfield’s semantics. The latter one proves that λ_i^{\dagger} ’s type system is at least as expressive as, if not stronger than, λ_i ’s.

Type Erasure. Differently from the other two systems, λ_i^{\dagger} uses type annotations in its syntax to obtain a direct operational semantics. $|e|$ erases annotations in term e . By erasing all annotations, terms in λ_i^{\dagger} can be converted to terms in Dunfield’s system and λ_i . The only exception is fixpoints, which λ_i does not have. The annotation erasure function is defined in Figure 6. Note that for every value v in λ_i^{\dagger} , $|v|$ is a value as well.

5.1 Soundness with respect to Dunfield’s Operational Semantics

Dunfield’s original reduction rules are presented in Fig 1. We extend her operational semantics with the following two rules. The full reduction rules can be found in the appendix.

$$\boxed{E \rightsquigarrow E'} \qquad \text{(The extension of Dunfield’s calculus)}$$

$$\begin{array}{c}
\text{DSTEP-TOP} \\
\hline
E \rightsquigarrow \top
\end{array}
\qquad
\begin{array}{c}
\text{DSTEP-TOPARR} \\
\hline
\top \rightsquigarrow \lambda x. \top
\end{array}$$

Rule DSTEP-TOP states that any value can be reduced to \top , corresponding to $A <: \text{Top}$. Rule DSTEP-TOPARR says that the value \top can be reduced to a lambda which returns \top , suggested by the subtyping rule S-TOPARR. Together with merge rules, the extended reduction can reduce any term to a value under a top-like type. Dunfield avoids having a rule DSTEP-TOP by performing a simplifying elaboration step advance:

$$\frac{}{\Gamma \vdash V : \text{Top} \hookrightarrow \top} \text{DUNFIELD-TYPING-T}$$

With such a rule, values of type Top are directly translated into \top , and do not need any further reduction in the target language. Accordingly, in her source language, there is no rule to convert these values to \top . We do not have such an elaboration step and we have already added rule DSTEP-TOPARR, so instead, we extend the original semantics with the two rules.

Soundness. Given Dunfield’s extended semantics, we can show a theorem that each step in the TDOS of λ_i corresponds to zero, one, or multiple steps in Dunfield’s semantics.

► **Theorem 18** (Soundness of \hookrightarrow with respect to Dunfield’s semantics). *If $e \hookrightarrow e'$, then $|e| \rightsquigarrow^* |e'|$.*

$$\boxed{\Gamma \models A} \qquad (Type\ wellformedness)$$

$$\begin{array}{c}
\text{WF-TOP} \\
\frac{}{\Gamma \models \text{Top}}
\end{array}
\qquad
\begin{array}{c}
\text{WF-INT} \\
\frac{}{\Gamma \models \text{Int}}
\end{array}
\qquad
\begin{array}{c}
\text{WF-ARR} \\
\frac{\Gamma \models A \quad \Gamma \models B}{\Gamma \models A \rightarrow B}
\end{array}
\qquad
\begin{array}{c}
\text{WF-AND} \\
\frac{\Gamma \models A \quad \Gamma \models B \quad A *_i B}{\Gamma \models A \& B}
\end{array}$$

$$\boxed{\Gamma \models E : A} \qquad (Typing)$$

$$\begin{array}{c}
\text{ITYP-TOP} \\
\frac{}{\Gamma \models \top : \text{Top}}
\end{array}
\qquad
\begin{array}{c}
\text{ITYP-LIT} \\
\frac{}{\Gamma \models i : \text{Int}}
\end{array}
\qquad
\begin{array}{c}
\text{ITYP-VAR} \\
\frac{x : A \in \Gamma}{\Gamma \models x : A}
\end{array}
\qquad
\begin{array}{c}
\text{ITYP-LAM} \\
\frac{\Gamma \models A \quad \Gamma, x : A \models E : B}{\Gamma \models (\lambda x. E) : A \rightarrow B}
\end{array}$$

$$\begin{array}{c}
\text{ITYP-APP} \\
\frac{\Gamma \models E_1 : A \rightarrow B \quad \Gamma \models E_2 : A}{\Gamma \models E_1 E_2 : B}
\end{array}
\qquad
\begin{array}{c}
\text{ITYP-MERGE} \\
\frac{\Gamma \models E_1 : A \quad \Gamma \models E_2 : B \quad A *_i B}{\Gamma \models E_1 , , E_2 : A \& B}
\end{array}
\qquad
\begin{array}{c}
\text{ITYP-SUB} \\
\frac{\Gamma \models E : A \quad A \triangleleft B}{\Gamma \models E : B}
\end{array}$$

■ **Figure 7** The declarative type system of λ_i .

A necessary lemma for this theorem is the soundness of typed reduction.

► **Lemma 19** (Soundness of Typed Reduction with respect to Dunfield's semantics). *If $v \hookrightarrow_A v'$, then $|v| \rightsquigarrow^* |v'|$.*

This lemma shows that although the type information guides the reduction of values, it does not add additional behavior to values. For example, a merge can step to its left part (or the right part) with rule `TREDUCE-MERGEVL` (or rule `TREDUCE-MERGEVR`), corresponding to rule `DSTEP-UNMERGEL` (or rule `DSTEP-UNMERGER`). And rule `TREDUCE-AND` ($v \hookrightarrow_{A \& B} v_1 , , v_2$ if $v \hookrightarrow_A v_1$ and $v \hookrightarrow_B v_2$) can be understood as a combination of splitting (rule `DSTEP-SPLIT` $V \rightsquigarrow V , , V$) and further reduction on each component separately.

In Section 6, we present another variant of $\lambda_i^{\dot{}}$, which has the same subtyping relation as Dunfield's system (minus union types). The same soundness theorem is proved for that variant without any modifications to Dunfield's operational semantics.

5.2 Completeness with respect to the Type System of λ_i

λ_i drops union types and introduces the disjointness restriction to Dunfield's system. When introducing λ_i , Oliveira et al. proposed an algorithmic and a declarative type system. The two type systems were shown to be equally expressive. For the declarative type system there is still the possibility of ambiguity due to the presence of an (implicit) subsumption rule (see also the discussion in Section 3.3). However, annotations in the bidirectional algorithmic type system ensure that well-typed terms in λ_i are unambiguous and subsumption is kept under control.

The type system of $\lambda_i^{\dot{}}$ is based on the declarative type system of λ_i , with three main changes:

1. λ_i^{\ddagger} forces the subsumption rule to be explicitly triggered by a type annotation.
2. λ_i^{\ddagger} supports fixpoints while λ_i does not.
3. λ_i^{\ddagger} has an additional rule for the merge of values (rule ETYP-MERGEV), which is required to prove type preservation, since duplicated values can occur in merges after reduction.

Some details need to be explained before presenting the completeness theorem. Firstly, because they are irrelevant, rules related to products and projection operators in λ_i are dropped. Secondly, the subtyping in λ_i^{\ddagger} is stronger due to the added rule S-TOPARR. Thirdly, top-like types are disjoint with any type in λ_i^{\ddagger} , while the disjointness in λ_i is restricted to types which are not top-like. The definition of λ_i^{\ddagger} 's subtyping and disjointness can be found in the appendix.

► **Theorem 20** (Completeness of Typing with respect to λ_i). *If $\Gamma \models E : A$, then there exists some e such that $\Gamma \vdash e : A$ and $E = |e|$.*

The above theorem shows that the type system of λ_i^{\ddagger} is able to type check any well-typed terms in λ_i , with proper type annotations inserted based on the typing derivation. It is built on the completeness of subtyping and disjointness of λ_i^{\ddagger} . The result means that λ_i 's type system (or any type system equivalent to it) can be used as a surface language where many of the explicit annotations of λ_i^{\ddagger} are inferred automatically. That is to say, the λ_i calculus can be translated without loss of expressivity or flexibility into λ_i^{\ddagger} .

To further show that some type inference with recursion is feasible, we extended the bi-directional type system of λ_i with recursion, and replaced the subtyping and disjointness by λ_i^{\ddagger} 's. We designed an elaboration from the extended system to λ_i^{\ddagger} and proved the following theorem. The typing rules can be found in the appendix.

► **Theorem 21** (Completeness of Typing with respect to the Extended Bidirectional Type System of λ_i). *If $\Gamma \vdash E \Rightarrow A \leftrightarrow e$ or $\Gamma \vdash E \Leftarrow A \leftrightarrow e$, then $\Gamma \vdash e : A$.*

6 Discussion

This section discusses one variant of λ_i^{\ddagger} , which is also formalized in Coq. The variant follows the subtyping relation in λ_i and Dunfield's calculus strictly and does not support multiple functions in merges. Some possible extensions to our work are also discussed.

6.1 A Variant of λ_i^{\ddagger}

In Section 5.1, we validate the TDOS of λ_i^{\ddagger} via a soundness theorem (Theorem 18) with respect to an extended operational semantics of Dunfield's calculus. In this section, we discuss a variant of λ_i^{\ddagger} that requires no extension to Dunfield's operational semantics. Its syntax and typing rules can be found in the appendix. Instead of adding rule S-TOPARR, this variant keeps the same subtyping relation as Dunfield's and adapts the definition of top-like types and disjointness, losing the ability to have multiple functions in a merge. Consequently, it is possible to prove the following soundness theorem on this variant without any modifications on Dunfield's operational semantics².

► **Theorem 22** (Soundness of \leftrightarrow in the simple variant). *If $e \leftrightarrow e'$, then $|e| \rightsquigarrow^* |e'|$.*

The above theorem states that each step taken by the TDOS corresponds to a series of reduction in the original operational semantics of Dunfield's calculus.

² For the syntax and rules of Dunfield's system, please refer to Section 2.

Besides soundness, this variant keeps the other important properties of λ_i : *determinism*, *type preservation* and *progress*. A completeness theorem with respect to the type system of λ_i is established as well.

► **Theorem 23** (Completeness of Typing in the simple variant). *If $\Gamma \models E : A$ in λ_i , then there exists some e such that $\Gamma \vdash e : A$ in the variant and $E = |e|$.*

Designing the Variant. As presented in Section 5.1, there are two reduction rules in λ_i that are related to the extension of Dunfield’s operational semantics: rule **TREDUCE-TOP** ($v \hookrightarrow_{\text{Top}} \top$) and rule **TREDUCE-TOPARR**. They reduce values under top-like types into a unified form. Without rule **S-TOPARR**, no arrow types are top-like, thus the latter is removed from the variant. However there is still rule **TREDUCE-TOP**, which is not accounted for in Dunfield’s original system. While we believe that such a rule fits in spirit well with the remaining non-deterministic rules, it is interesting to see if it is possible to model a calculus without it (and without extending Dunfield semantics at all).

Reducing a value v under type **Top**, in fact, can be thought as seeking an inhabited value of **Top** which acts like v . In Dunfield’s original semantics there is no way to convert a value to \top , which is our source of difficulties. Dunfield solves this problem by having a typing rule for values that allows any value to have type **Top**. This works well in her setting because she can have ambiguous terms. Unfortunately, it does not work well in our setting because, as discussed in detail in Section 3.3, allowing values to be implicitly typed as **Top** provides a way to bypass the disjointness restrictions. We overcome the problem instead by introducing a new value construct $v : \text{Top}$ in our variant. This new form of value inhabits the **Top** type but, unlike \top it does not forget about the original value v (which can be of any type). Thus the original value v , can be recovered by erasing the wrapped annotation.

$$\frac{\text{TRED-TOP}}{v \hookrightarrow_{\text{Top}} v : \text{Top}}$$

Although the new rule then corresponds to $v \rightsquigarrow v$ after annotation erasure, it breaks determinism as a merge can reduce to either its left or right component, leading to different results, e.g. $1, , \text{True} \hookrightarrow_{\text{Top}} 1 : \text{Top}$ and $1, , \text{True} \hookrightarrow_{\text{Top}} \text{True} : \text{Top}$. To solve this problem we directly reduce the value before splitting merges by excluding **Top** from ordinary types.

To use the new construct for expressions and mix it with annotated terms, values and expressions are separated into two syntactic categories in the variant (but all values can be treated as expressions $\langle v \rangle$). The partition results in some tedious rules in the reduction relation. For instance, $\langle v_1 \rangle, , \langle v_2 \rangle \hookrightarrow \langle v_1, , v_2 \rangle$ reduces a merge of two values to a merged value.

6.2 Improvements and Extensions

Less Checks on Reduction. In rule **TREDUCE-ARROW** (in Figure 4), the premise $C <: A$ is actually redundant for the purposes of reduction. Since we only care about well-typed terms being reduced, such a check has already been guaranteed by typing. Therefore an actual implementation could omit that check. The reason why we keep the premise is that typed reduction plays another role in our metatheory: it allows us to define consistency. Consistency is defined for any (untyped) values, and the extra check there tightens up the definition of consistency. With the premise, typed reduction directly implies a subtyping relation between the principal type of the reduced value and the reduction type. (See Lemma 9: If $v \hookrightarrow_A v'$, then $\text{type}_p \langle v \rangle <: A$). One could wonder if this property is unnecessary because it may be

derived by type preservation of reduction. Note that whenever typed reduction is called in a reduction rule, the subtyping relation can be obtained from the typing derivation of the reduced term. For example, reducing $v : A$ will type reduce v under A . If $v : A$ is well-typed, then we could in principle prove that $\text{type}_p\langle v \rangle <: A$. Unfortunately, the above proof is hard to attain in practice. Because type preservation depends on consistency, and consistency is defined by typed reduction. Once the subtyping property relies on type preservation, there is a cyclic dependency between the properties. In future work we would like to look at this issue more closely and try to discard the premise by taking full advantage of the type system.

Distributive Subtyping. Although the subtyping of $\lambda_i^{\dot{}}$ allows multiple functions in a merge, it lacks the distributive subtyping rule for intersection types that has been employed in some recent calculi [5,6]. The distributivity of intersections over arrows $((A \rightarrow B_1) \& (A \rightarrow B_2) <: A \rightarrow B_1 \& B_2)$ [3] is well accepted for its theoretical elegance. But it is also well-known for being troublesome. Mainly, there are two challenges for adapting distributive subtyping to $\lambda_i^{\dot{}}$.

- The rule indicates that a merge of functions can be applied. While the current typing rule can check such application with suitable annotations, designing new reduction rules is necessary. A promising solution is to have a rule allows parallel application like $(v_1 \ , \ v_2) v \hookrightarrow v_1 v \ , \ v_2 v$.
- Function types are no longer “ordinary”. In $\lambda_i^{\dot{}}$, the intuition behind ordinary types is that their typed reduction results never contains merges, which is necessary for determinism. With distributivity, typed reduction may produce a merge under a single function type. For example, $\lambda x. 'c' : \text{Int} \rightarrow \text{Char} \ , \ \lambda x. x : \text{Int} \rightarrow \text{Int} \hookrightarrow_{\text{Int} \rightarrow \text{Char} \& \text{Int}} \lambda x. 'c' : \text{Int} \rightarrow \text{Char} \ , \ \lambda x. x : \text{Int} \rightarrow \text{Int}$. In the typed reduction of $\lambda_i^{\dot{}}$, intersections are split into basic units. However, it is not straightforward to split a function type.

7 Related Work

7.1 Calculi with the Merge Operator and a Direct Semantics

Intersection types with a merge operator are a key feature of Reynolds’ Forsythe language [43]. Reynolds studied a core calculus [43] with similarities to $\lambda_i^{\dot{}}$. However, merges in Forsythe are restricted and use a syntactic criterion to determine what merges are allowed. A merge is permitted only when the second term is a lambda abstraction or a single field record, which makes the structure of merge always biased. To prevent potential ambiguity, the latter overrides the former when overlapped. Note that the structure of merge in Forsythe is always biased. If formalized as a tree, the right child of every node is a leaf. The only place for primitive types is the leftmost component. Forsythe follows the standard call-by-name small-step reduction, during which types are ignored. The reduction rules deal with merges by continuously checking if the second component can be used in the context (abstractions for application, records for projection). This simple approach, however, is unable to reduce merges when (multiple) primitive types are required. Reynolds admits this issue in his later work [45]. In $\lambda_i^{\dot{}}$ types are used to select values from a merge and the disjointness restriction guarantees the determinism. Therefore the order of a value in a merge is not a deciding factor on whether the value is used or not.

The calculus $\lambda\&$ proposed by Castagna et al. [9] has a restricted version of the merge operator for functions only. The merge operator is indexed by a list of types of its components. The operational semantics uses the run-time types of values to select the “best approximate” branch of an overloaded function. $\lambda\&$ requires run-time type checking on values, while

	Dunfield's [23]	λ_i [20]	F_i [2]	NeColus [5]	F_i^+ [6]	λ_i^\dagger
Disjointness	○	●	●	●	●	●
Unrestricted Intersections	●	○	○	●	●	●
Determinism or Coherence	No	Coh.	Coh.	Coh.	Coh.	Det.
Coercion Free	●	○	○	○	○	●
Recursion	●	○	○	○	○	●
Direct Semantics	●	○	○	○	○	●
Subject-Reduction	○	-	-	-	-	●

■ **Figure 8** Summary of intersection calculi with the merge operator (● = yes, ○ = no, - = not applicable).

in TDOS, all type information is present already in type annotations. Another obvious difference is that λ_i^\dagger supports merges of any types (not just functions), which are useful for applications other than overloading of functions, including: multifield *extensible records with subtyping* [20]; encodings of *objects* and *traits* [4]; *dynamic mixins* [2]; or simple forms of *family polymorphism* [5].

Several other calculi with intersection types and overloading of functions have been proposed [10–12], but these calculi do not support a merge operator, and thus avoid the ambiguity problems caused by the construct.

7.2 Calculi with a Merge Operator and an Elaboration Semantics

Instead of a direct semantics, many recent works [2, 5, 6, 20, 23] on intersection types employ an elaboration semantics, translating merges in the source language to products (or pairs) in a target language. With an elaboration semantics the subtyping derivations are coercive [33]: they produce coercion functions that explicitly convert terms of one type to another in the target language. This idea was first proposed by Dunfield [23], where she shows how to elaborate a calculus with intersection and union types and a merge operator to a standard call-by-value lambda calculus with products and sums. Dunfield also proposed a direct semantics, which served as inspiration for our own work. However, her direct semantics is non-deterministic and lacks subject-reduction (as discussed in detail in Section 2.2). Unlike Forsythe and $\lambda\&$, Dunfield’s calculus has unrestricted merges and allows a merge to work as an argument. Her calculus is flexible and expressive and can deal with several programs that are not allowed in Forsythe and $\lambda\&$.

To remove the ambiguity issues in Dunfield’s work, the λ_i calculus [20] forbids overlapping in intersections using the disjointness restriction for all well-formed intersections. In other words, λ_i does not support unrestricted intersections. Because of this restriction, the proof of coherence in λ_i is still relatively simple. Likewise, in following work on the F_i calculus [2], which extends λ_i with disjoint polymorphism, *all* intersections must be *disjoint*. However the disjointness restriction causes difficulties because it breaks *stability of type substitutions*. Stability is a desirable property in a polymorphic type system that ensures that if a polymorphic type is well-formed then any instantiation of that type is also well-formed. Unfortunately, with disjoint intersections only, this property is not true in general. Thus F_i can only prove a restricted version of stability, which makes its metatheory non-trivial.

Disjointness of all well-formed intersections is only a sufficient (but not necessary) restriction to ensure an unambiguous semantics. The NeColus calculus [5] relaxes the restriction without introducing ambiguity. In NeColus 1 : $\text{Int} \& \text{Int}$ is allowed, but the same term is

rejected in λ_i . In other words, NeColus employs the disjointness restriction *only* on merges, but otherwise allows *unrestricted intersections*. Unfortunately, this comes at a cost: it is much harder to prove the coherence of elaboration. Both NeColus and F_i^+ [6] (a calculus derived from F_i that allows unrestricted intersections) deal with this problem by establishing coherence using contextual equivalence and a *logical relation* [40, 49, 50] to prove it. The proof method, however, cannot deal with non-terminating programs. In fact none of the existing calculi with disjoint intersection types supports recursion, which is a severe restriction.

We retain the essence of the power of Dunfield’s calculus (modulo the disjointness restrictions to rule out ambiguity), and gain benefits from the direct semantics. Figure 8 summarizes the key differences between our work and prior work, focusing on the most recent work on disjoint intersection types. Note that the row titled “Coercion Free” denotes whether subtyping generates coercions or not. λ_i^* is coercion free, while all other calculi based on an elaboration semantics employ coercive subtyping. Next we give more detail on the advantages of a direct semantics over the elaboration semantics and proof methods employed in previous work on disjoint intersection types.

Shorter, more Direct Reasoning. Programmers want to understand the meaning of their programs. A formal semantics can help with this. With our TDOS semantics we can essentially employ a style similar to equational reasoning in functional programming to directly reason about programs written in λ_i^* . For example, it takes a few reasoning steps to work out the result of $(\lambda x. x + 1 : \text{Int} \rightarrow \text{Int}) (2, , 'c')$:

$$\begin{array}{ll}
 (\lambda x. x + 1 : \text{Int} \rightarrow \text{Int}) (2, , 'c') & \\
 \hookrightarrow (2 + 1) : \text{Int} & \{\text{by STEP-BETA and typed reduction of argument under Int}\} \\
 \hookrightarrow 3 : \text{Int} & \{\text{by STEP-ANNO and usual reduction rules for arithmetic}\} \\
 \hookrightarrow 3 & \{\text{by STEP-ANNOV and typed reduction of 3 under Int}\}
 \end{array}$$

Here reasoning is easily justifiable from the small-step reduction rules and type-directed reduction. In fact building tools (such as some form of debugger), that automate such kind of reasoning should be easy using the TDOS rules.

However, with an elaboration semantics, the (precise) reasoning steps to determine the final result are much more complex. Firstly the expression has to be translated into the target language before reducing to a similar target term. Figure 9 shows this elaboration process in λ_i , where an expression in the source language is translated into an expression in a target language with products. The source term $(\lambda x. x + 1 : \text{Int} \rightarrow \text{Int}) (2, , 'c')$ is elaborated into the target term $(\lambda x. x + 1) (\text{fst } (2, 'c'))$. As we can see the actual derivation is rather long, so we skip the full steps. Also, for simplicity’s sake, here we assume the subtyping judgement produces the most straightforward coercion `fst`. This elaboration step together with the introduction of coercions into the program makes it much harder for programmers to precisely understand the semantics of a program. Moreover while the coercions inserted in this small expression may not look too bad, in larger programs the addition of coercions can be a lot more severe, hampering the understanding of the program. After elaboration we can then use the target language semantics, to determine a target language value.

$$\begin{array}{ll}
 (\lambda x. x + 1) (\text{fst } (2, 'c')) & \\
 \hookrightarrow (\lambda x. x + 1) 2 & \{\text{by a rule similar to STEP-APPR and reduction rules for pairs}\} \\
 \hookrightarrow 2 + 1 & \{\text{by beta reduction rule}\} \\
 \hookrightarrow 3 & \{\text{by usual reduction rules for arithmetic}\}
 \end{array}$$

$$\begin{array}{c}
\text{T-MERGE} \\
\frac{\dots}{\cdot \vdash (2, , 'c') \Rightarrow \text{Int} \& \text{Char} \rightsquigarrow (2, 'c')} \\
\text{T-ANN} \\
\frac{\dots}{\cdot \vdash (\lambda x. x + 1 : \text{Int} \rightarrow \text{Int}) \Rightarrow \text{Int} \rightarrow \text{Int} \rightsquigarrow \lambda x. x + 1} \\
\vdots \\
\text{T-SUB} \frac{\text{SUB-ANDL} \quad \text{Int} \& \text{Char} <: \text{Int} \rightsquigarrow \text{fst}}{\cdot \vdash (2, , 'c') \Leftarrow \text{Int} \rightsquigarrow \text{fst} (2, 'c')} \\
\text{T-APP} \frac{\dots}{\cdot \vdash (\lambda x. x + 1 : \text{Int} \rightarrow \text{Int}) (2, , 'c') \Rightarrow \text{Int} \rightsquigarrow (\lambda x. x + 1) (\text{fst} (2, 'c'))}
\end{array}$$

■ **Figure 9** Elaboration of the expression $(\lambda x. x + 1 : \text{Int} \rightarrow \text{Int}) (2, , 'c')$ into a target calculus with products.

A final issue is that sometimes it is not even possible to translate back the value of the target language into an equivalent “value” on the source. For instance in the NeColus calculus [5] $1 : \text{Int} \& \text{Int}$ results in $(1, 1)$, which is a pair in the target language. But the corresponding source value $1, , 1$ is not typable in NeColus. In essence, with an elaboration, programmers must understand not only the source language, but also the elaboration process as well as the semantics of the target language, if they want to precisely understand the semantics of a program. Since the main point of semantics is to give clear and simple rules to understand the meaning of programs, a direct semantics is a better option for providing such understanding.

Simpler Proofs of Unambiguity. For calculi with an elaboration semantics, unrestricted intersections make it harder to prove the coherence. Our λ_i^c calculus, on the other hand, has a deterministic semantics, which implies unambiguity directly. For instance, $(1 : \text{Int} \& \text{Int}) : \text{Int}$ only steps to 1 in λ_i^c . But it can be elaborated into two target expressions in the NeColus calculus corresponding to two typing derivations:

$$(1 : \text{Int} \& \text{Int}) : \text{Int} \rightsquigarrow \text{fst} (1, 1)$$

$$(1 : \text{Int} \& \text{Int}) : \text{Int} \rightsquigarrow \text{snd} (1, 1)$$

Thus the coherence proof needs deeper knowledge about the semantics: the two different terms are known to both reduce to 1 eventually. Therefore they are related by the logical relation employed in NeColus for coherence. Things get more complicated for functions. The following example shows two possible elaborations of the same function. To relate them requires reasoning inside the binders and a notion of contextual equivalence.

$$\lambda x. x + 1 : \text{Int} \& \text{Int} \rightarrow \text{Int} \rightsquigarrow \lambda x. \text{fst } x + 1$$

$$\lambda x. x + 1 : \text{Int} \& \text{Int} \rightarrow \text{Int} \rightsquigarrow \lambda x. \text{snd } x + 1$$

Furthermore, the two target expressions above are *clearly not equivalent* in the general case. For instance, if we apply them to $(1, 2)$ we get different results. However, the target expressions will always behave equivalently when applied to arguments *elaborated from the NeColus source calculus*. NeColus, forbids terms like $(1, , 2)$ and thus cannot produce a target value $(1, 2)$. Because of elaboration and also this deeper form of reasoning required to show the equivalence of semantics, calculi defined by elaboration require a lot more infrastructure for the source and target calculi and the elaboration between them, while in a direct semantics only one calculus is involved and the reasoning required to prove determinism is quite simple.

Not Limited to Terminating Programs. The (basic) forms of logical relations employed by NeColus and F_i^+ has cannot deal with non-terminating programs. In principle, recursion could be supported by using a *step-indexed logical relation* [1], but this is left for future work (and it is non-trivial). $\lambda_i^{\dot{}}$ smoothly handles unrestricted intersections and recursion, using TDOS to reach determinism with a significantly simpler proof method. It also makes other features that lead to non-terminating programs, such as *recursive types*, feasible.

7.3 Languages and Calculi with Type-Dependent Semantics

Typed Operational Semantics Goguen [29] uses types in its reduction judgment, similarly to typed reduction in $\lambda_i^{\dot{}}$. However, Goguen’s typed operational semantics is designed for studying meta-theoretic properties, especially strong normalization, and is not aimed to describe type-dependent semantics. Unlike TDOS, in typed operational semantics the reduction process does not use the additional type information to guide reduction. Instead, the combination of well-typedness and computation provides inversion principles for proving various metatheoretical properties. Typed operational semantics has been applied to several systems. These include *simply typed lambda calculi* [30], calculi with *dependent types* [26, 29] and *higher-order subtyping* [17]. Note that the semantics of these systems does not depend on typing, and the untyped (type-erased) reduction relations are still presented to describe how to evaluate programs.

Type classes [32, 52] are an approach to parametric overloading used in languages like Haskell. The commonly adopted compilation strategy for it is the dictionary passing style elaboration [13, 14, 31, 52]. Other mechanisms inspired by type classes, such as Scala’s *implicit*s [19], Agda’s *instance arguments* [22] or Ocaml’s *modular implicit*s [54] have an elaboration semantics as well. In one of the pioneering works of type classes, Kaes [32] gives two formulations for a direct operational semantics. One of them decides the concrete type of the instance of overloaded functions at *run-time*, by analyzing all arguments after evaluating them. In both Kaes’ work and a following work by Odersky et al. [37], the run-time semantics has some restrictions with respect to type classes. For example, overloading on return types (needed for example for the *read* function in Haskell) is not supported. Interestingly, the semantics of $\lambda_i^{\dot{}}$ allows overloading on return types, which is used whenever two functions coexist on a merge.

Gradual typing [47] has become popular over the last few years. Gradual typing is another example of a type-dependent mechanism, since the success or not of an (implicit) cast may depend on the particular type used for the implicit cast. Thus the semantics of a gradually typed language is type-dependent. Like other type-dependent mechanisms the semantics of gradually typed source languages is usually given by a (type-dependent) elaboration semantics into a cast calculus, such as the *Blame calculus* [53] or the *Threesome calculus* [48].

Static binding of fields and *method overloading* in Java [51] make use of type annotations computed in a preprocessing phase. For each method invocation, the annotation states the argument type of the most specific method applicable according to the static types. Based on the annotation and the run-time type (class) of the object, a dynamic lookup function yields a proper method at run-time. This allows static method overloading works across the inheritance hierarchy, together with dynamic dispatch. *Multiple dispatching* [15, 16, 35, 38] generalizes object-oriented dynamic dispatch to determine the overloaded method to invoke based on the run-time type of all its arguments. Similarly to TDOS, much of the type information is recovered from type annotations in multiple dispatching mechanisms, but, unlike TDOS, they only use input types to determine the semantics.

8 Conclusion

In this work we presented a TDOS for $\lambda_i^{\dot{}}$: a calculus that includes intersection types and an expressive unbiased merge operator. Among all similar calculi, $\lambda_i^{\dot{}}$ is the first to have a direct operational semantics that is both deterministic and has subject-reduction. Compared with the elaboration approach, having a direct semantics avoids the translation process and a target calculus. This simplifies both informal and formal reasoning. For instance, establishing the coherence of elaboration in NeColus [5] requires much more sophistication than obtaining the determinism theorem in $\lambda_i^{\dot{}}$. Furthermore the proof method for coherence in NeColus cannot deal with non-terminating programs, whereas dealing with recursion is straightforward in $\lambda_i^{\dot{}}$. The semantics of $\lambda_i^{\dot{}}$ exploits type annotations to guide reduction. The key component of TDOS is *typed reduction*, which allows values to be further reduced depending on their type. For the future we would like to develop further the TDOS approach in the setting of disjoint intersection types. Some interesting extensions include support for *distributive subtyping* [3], *disjoint polymorphism* [2] and iso-recursive types with the Amber rule [7].

References

- 1 Amal J. Ahmed. Step-indexed syntactic logical relations for recursive and quantified types. In Peter Sestoft, editor, *Programming Languages and Systems, 15th European Symposium on Programming, ESOP 2006, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2006, Vienna, Austria, March 27-28, 2006, Proceedings*, volume 3924 of *Lecture Notes in Computer Science*, pages 69–83. Springer, 2006. doi:10.1007/11693024_6.
- 2 João Alpuim, Bruno C. d. S. Oliveira, and Zhiyuan Shi. Disjoint polymorphism. In Hongseok Yang, editor, *Programming Languages and Systems - 26th European Symposium on Programming, ESOP 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings*, volume 10201 of *Lecture Notes in Computer Science*, pages 1–28. Springer, 2017. doi:10.1007/978-3-662-54434-1_1.
- 3 Henk Barendregt, Mario Coppo, and Mariangiola Dezani-Ciancaglini. A filter lambda model and the completeness of type assignment 1. *The journal of symbolic logic*, 48(4), 1983.
- 4 Xuan Bi and Bruno C. d. S. Oliveira. Typed first-class traits. In Todd D. Millstein, editor, *32nd European Conference on Object-Oriented Programming, ECOOP 2018, July 16-21, 2018, Amsterdam, The Netherlands*, volume 109 of *LIPICs*, pages 9:1–9:28. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2018. doi:10.4230/LIPICs.ECOOP.2018.9.
- 5 Xuan Bi, Bruno C. d. S. Oliveira, and Tom Schrijvers. The essence of nested composition. In Todd D. Millstein, editor, *32nd European Conference on Object-Oriented Programming, ECOOP 2018, July 16-21, 2018, Amsterdam, The Netherlands*, volume 109 of *LIPICs*, pages 22:1–22:33. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2018. doi:10.4230/LIPICs.ECOOP.2018.22.
- 6 Xuan Bi, Ningning Xie, Bruno C. d. S. Oliveira, and Tom Schrijvers. Distributive disjoint polymorphism for compositional programming. In Luís Caires, editor, *Programming Languages and Systems - 28th European Symposium on Programming, ESOP 2019, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2019, Prague, Czech Republic, April 6-11, 2019, Proceedings*, volume 11423 of *Lecture Notes in Computer Science*, pages 381–409. Springer, 2019. doi:10.1007/978-3-030-17184-1_14.
- 7 Luca Cardelli. Amber. In Guy Cousineau, Pierre-Louis Curien, and Bernard Robinet, editors, *Combinators and Functional Programming Languages, Thirteenth Spring School of the LITP, Val d'Ajol, France, May 6-10, 1985, Proceedings*, volume 242 of *Lecture Notes in Computer Science*, pages 21–47. Springer, 1985. doi:10.1007/3-540-17184-3_38.

- 8 Luca Cardelli and Peter Wegner. On understanding types, data abstraction, and polymorphism. *ACM Comput. Surv.*, 17(4):471–522, 1985. doi:10.1145/6041.6042.
- 9 Giuseppe Castagna, Giorgio Ghelli, and Giuseppe Longo. A calculus for overloaded functions with subtyping. *Inf. Comput.*, 117(1):115–135, 1995. doi:10.1006/inco.1995.1033.
- 10 Giuseppe Castagna, Kim Nguyen, Zhiwu Xu, and Pietro Abate. Polymorphic functions with set-theoretic types: Part 2: Local type inference and type reconstruction. In Sriram K. Rajamani and David Walker, editors, *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, Mumbai, India, January 15-17, 2015*, pages 289–302. ACM, 2015. doi:10.1145/2676726.2676991.
- 11 Giuseppe Castagna, Kim Nguyen, Zhiwu Xu, Hyeonseung Im, Sergueï Lenglet, and Luca Padovani. Polymorphic functions with set-theoretic types: part 1: syntax, semantics, and evaluation. In Suresh Jagannathan and Peter Sewell, editors, *The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '14, San Diego, CA, USA, January 20-21, 2014*, pages 5–18. ACM, 2014. doi:10.1145/2535838.2535840.
- 12 Giuseppe Castagna and Zhiwu Xu. Set-theoretic foundation of parametric polymorphism and subtyping. In Manuel M. T. Chakravarty, Zhenjiang Hu, and Olivier Danvy, editors, *Proceeding of the 16th ACM SIGPLAN international conference on Functional Programming, ICFP 2011, Tokyo, Japan, September 19-21, 2011*, pages 94–106. ACM, 2011. doi:10.1145/2034773.2034788.
- 13 Manuel M. T. Chakravarty, Gabriele Keller, and Simon L. Peyton Jones. Associated type synonyms. In Olivier Danvy and Benjamin C. Pierce, editors, *Proceedings of the 10th ACM SIGPLAN International Conference on Functional Programming, ICFP 2005, Tallinn, Estonia, September 26-28, 2005*, pages 241–253. ACM, 2005. doi:10.1145/1086365.1086397.
- 14 Manuel M. T. Chakravarty, Gabriele Keller, Simon L. Peyton Jones, and Simon Marlow. Associated types with class. In Jens Palsberg and Martín Abadi, editors, *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2005, Long Beach, California, USA, January 12-14, 2005*, pages 1–13. ACM, 2005. doi:10.1145/1040305.1040306.
- 15 Craig Chambers and Weimin Chen. Efficient multiple and predicated dispatching. In Brent Hailpern, Linda M. Northrop, and A. Michael Berman, editors, *Proceedings of the 1999 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages & Applications (OOPSLA '99), Denver, Colorado, USA, November 1-5, 1999*, pages 238–255. ACM, 1999. doi:10.1145/320384.320407.
- 16 Curtis Clifton, Gary T. Leavens, Craig Chambers, and Todd D. Millstein. Multijava: modular open classes and symmetric multiple dispatch for java. In Mary Beth Rosson and Doug Lea, editors, *Proceedings of the 2000 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages & Applications (OOPSLA 2000), Minneapolis, Minnesota, USA, October 15-19, 2000*, pages 130–145. ACM, 2000. doi:10.1145/353171.353181.
- 17 Adriana B. Compagnoni and Healfdene Goguen. Typed operational semantics for higher-order subtyping. *Inf. Comput.*, 184(2):242–297, 2003. doi:10.1016/S0890-5401(03)00062-2.
- 18 Mario Coppo, Mariangiola Dezani-Ciancaglini, and Betti Venneri. Functional characters of solvable terms. *Math. Log. Q.*, 27(2-6):45–58, 1981. doi:10.1002/malq.19810270205.
- 19 Bruno C. d. S. Oliveira, Adriaan Moors, and Martin Odersky. Type classes as objects and implicits. In William R. Cook, Siobhán Clarke, and Martin C. Rinard, editors, *Proceedings of the 25th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2010, October 17-21, 2010, Reno/Tahoe, Nevada, USA*, pages 341–360. ACM, 2010. doi:10.1145/1869459.1869489.
- 20 Bruno C. d. S. Oliveira, Zhiyuan Shi, and João Alpuim. Disjoint intersection types. In Jacques Garrigue, Gabriele Keller, and Eijiro Sumii, editors, *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming, ICFP 2016, Nara, Japan, September 18-22, 2016*, pages 364–377. ACM, 2016. doi:10.1145/2951913.2951945.

- 21 Rowan Davies and Frank Pfenning. Intersection types and computational effects. In Martin Odersky and Philip Wadler, editors, *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming (ICFP '00), Montreal, Canada, September 18-21, 2000*, pages 198–208. ACM, 2000. doi:10.1145/351240.351259.
- 22 Dominique Devriese and Frank Piessens. On the bright side of type classes: instance arguments in agda. In Manuel M. T. Chakravarty, Zhenjiang Hu, and Olivier Danvy, editors, *Proceeding of the 16th ACM SIGPLAN international conference on Functional Programming, ICFP 2011, Tokyo, Japan, September 19-21, 2011*, pages 143–155. ACM, 2011. doi:10.1145/2034773.2034796.
- 23 Jana Dunfield. Elaborating intersection and union types. *J. Funct. Program.*, 24(2-3):133–165, 2014. doi:10.1017/S0956796813000270.
- 24 Jana Dunfield and Frank Pfenning. Type assignment for intersections and unions in call-by-value languages. In Andrew D. Gordon, editor, *Foundations of Software Science and Computational Structures, 6th International Conference, FOSSACS 2003 Held as Part of the Joint European Conference on Theory and Practice of Software, ETAPS 2003, Warsaw, Poland, April 7-11, 2003, Proceedings*, volume 2620 of *Lecture Notes in Computer Science*, pages 250–266. Springer, 2003. doi:10.1007/3-540-36576-1_16.
- 25 Facebook. Flow. <https://flow.org/>, 2014.
- 26 Yangyue Feng and Zhaohui Luo. Typed operational semantics for dependent record types. In Tom Hirschowitz, editor, *Proceedings Types for Proofs and Programs, Revised Selected Papers, TYPES 2009, Aussois, France, 12-15th May 2009*, volume 53 of *EPTCS*, pages 30–46, 2009. doi:10.4204/EPTCS.53.3.
- 27 Matthew Flatt, Shriram Krishnamurthi, and Matthias Felleisen. Classes and mixins. In David B. MacQueen and Luca Cardelli, editors, *POPL '98, Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Diego, CA, USA, January 19-21, 1998*, pages 171–183. ACM, 1998. doi:10.1145/268946.268961.
- 28 Timothy S. Freeman and Frank Pfenning. Refinement types for ML. In David S. Wise, editor, *Proceedings of the ACM SIGPLAN'91 Conference on Programming Language Design and Implementation (PLDI), Toronto, Ontario, Canada, June 26-28, 1991*, pages 268–277. ACM, 1991. doi:10.1145/113445.113468.
- 29 Healfdene Goguen. *A typed operational semantics for type theory*. PhD thesis, University of Edinburgh, UK, 1994. URL: <http://hdl.handle.net/1842/405>.
- 30 Healfdene Goguen. Typed operational semantics. In Mariangiola Dezani-Ciancaglini and Gordon D. Plotkin, editors, *Typed Lambda Calculi and Applications, Second International Conference on Typed Lambda Calculi and Applications, TLCA '95, Edinburgh, UK, April 10-12, 1995, Proceedings*, volume 902 of *Lecture Notes in Computer Science*, pages 186–200. Springer, 1995. doi:10.1007/BFb0014053.
- 31 Cordelia V. Hall, Kevin Hammond, Simon L. Peyton Jones, and Philip Wadler. Type classes in haskell. *ACM Trans. Program. Lang. Syst.*, 18(2):109–138, 1996. doi:10.1145/227699.227700.
- 32 Stefan Kaes. Parametric overloading in polymorphic programming languages. In Harald Ganzinger, editor, *ESOP '88, 2nd European Symposium on Programming, Nancy, France, March 21-24, 1988, Proceedings*, volume 300 of *Lecture Notes in Computer Science*, pages 131–144. Springer, 1988. doi:10.1007/3-540-19027-9_9.
- 33 Zhaohui Luo. Coercive subtyping. *J. Log. Comput.*, 9(1):105–130, 1999. doi:10.1093/logcom/9.1.105.
- 34 Microsoft. Typescript. <https://www.typescriptlang.org/>, 2012.
- 35 Radu Muschevici, Alex Potanin, Ewan D. Tempero, and James Noble. Multiple dispatch in practice. 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 563–582. ACM, 2008. doi:10.1145/1449764.1449808.

- 36 Martin Odersky, Philippe Altherr, Vincent Cremet, Burak Emir, Sebastian Maneth, Stéphane Micheloud, Nikolay Mihaylov, Michel Schinz, Erik Stenman, and Matthias Zenger. An overview of the scala programming language. Technical report, École Polytechnique Fédérale de Lausanne, 2004.
- 37 Martin Odersky, Philip Wadler, and Martin Wehr. A second look at overloading. In John Williams, editor, *Proceedings of the seventh international conference on Functional programming languages and computer architecture, FPCA 1995, La Jolla, California, USA, June 25-28, 1995*, pages 135–146. ACM, 1995. doi:10.1145/224164.224195.
- 38 Gyunghye Park, Jaemin Hong, Guy L. Steele Jr., and Sukyoung Ryu. Polymorphic symmetric multiple dispatch with variance. *Proc. ACM Program. Lang.*, 3(POPL):11:1–11:28, 2019. doi:10.1145/3290324.
- 39 Benjamin C. Pierce. *Programming with Intersection Types and Bounded Polymorphism*. PhD thesis, Carnegie Mellon University, December 1991.
- 40 Gordon Plotkin. Lambda-definability and logical relations, 1973.
- 41 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.
- 42 Redhat. Ceylon. <https://ceylon-lang.org/>, 2011.
- 43 John C Reynolds. Preliminary design of the programming language Forsythe. Technical Report CMU-CS-88-159, Carnegie Mellon University, 1988.
- 44 John C. Reynolds. The coherence of languages with intersection types. In Takayasu Ito and Albert R. Meyer, editors, *Theoretical Aspects of Computer Software, International Conference TACS '91, Sendai, Japan, September 24-27, 1991, Proceedings*, volume 526 of *Lecture Notes in Computer Science*, pages 675–700. Springer, 1991. doi:10.1007/3-540-54415-1_70.
- 45 John C Reynolds. Design of the programming language Forsythe. In *ALGOL-like languages*, pages 173–233. Springer, 1997.
- 46 Nathanael Schärli, Stéphane Ducasse, Oscar Nierstrasz, and Andrew P. Black. Traits: Composable units of behaviour. In Luca Cardelli, editor, *ECOOP 2003 - Object-Oriented Programming, 17th European Conference, Darmstadt, Germany, July 21-25, 2003, Proceedings*, volume 2743 of *Lecture Notes in Computer Science*, pages 248–274. Springer, 2003. doi:10.1007/978-3-540-45070-2_12.
- 47 Jeremy G Siek and Walid Taha. Gradual typing for functional languages. In *Scheme and Functional Programming Workshop*, 2006.
- 48 Jeremy G. Siek and Philip Wadler. Threesomes, with and without blame. In Manuel V. Hermenegildo and Jens Palsberg, editors, *Proceedings of the 37th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2010, Madrid, Spain, January 17-23, 2010*, pages 365–376. ACM, 2010. doi:10.1145/1706299.1706342.
- 49 Richard Statman. Logical relations and the typed λ -calculus. *Inf. Control.*, 65(2/3):85–97, 1985. doi:10.1016/S0019-9958(85)80001-2.
- 50 William W. Tait. Intensional interpretations of functionals of finite type I. *J. Symb. Log.*, 32(2):198–212, 1967. doi:10.2307/2271658.
- 51 David von Oheimb and Tobias Nipkow. Machine-checking the java specification: Proving type-safety. In Jim Alves-Foss, editor, *Formal Syntax and Semantics of Java*, volume 1523 of *Lecture Notes in Computer Science*, pages 119–156. Springer, 1999. doi:10.1007/3-540-48737-9_4.
- 52 Philip Wadler and Stephen Blott. How to make ad-hoc polymorphism less ad-hoc. In *Conference Record of the Sixteenth Annual ACM Symposium on Principles of Programming Languages, Austin, Texas, USA, January 11-13, 1989*, pages 60–76. ACM Press, 1989. doi:10.1145/75277.75283.
- 53 Philip Wadler and Robert Bruce Findler. Well-typed programs can't be blamed. In Giuseppe Castagna, editor, *Programming Languages and Systems, 18th European Symposium on Programming, ESOP 2009, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009, York, UK, March 22-29, 2009. Proceedings*, volume 5502 of *Lecture Notes in Computer Science*, pages 1–16. Springer, 2009. doi:10.1007/978-3-642-00590-9_1.

26:32 A Type-Directed Operational Semantics for a Calculus with a Merge Operator

- 54 Leo White, Frédéric Bour, and Jeremy Yallop. Modular implicits. In Oleg Kiselyov and Jacques Garrigue, editors, *Proceedings ML Family/OCaml Users and Developers workshops, ML/OCaml 2014, Gothenburg, Sweden, September 4-5, 2014*, volume 198 of *EPTCS*, pages 22–63, 2014. doi:10.4204/EPTCS.198.2.
- 55 Andrew K. Wright and Matthias Felleisen. A syntactic approach to type soundness. *Inf. Comput.*, 115(1):38–94, 1994. doi:10.1006/inco.1994.1093.

Row and Bounded Polymorphism via Disjoint Polymorphism

Ningning Xie


The University of Hong Kong, China
nnxie@cs.hku.hk

Bruno C. d. S. Oliveira

The University of Hong Kong, China
bruno@cs.hku.hk

Xuan Bi

The University of Hong Kong, China
xbi@cs.hku.hk

Tom Schrijvers 

KU Leuven, Belgium
<https://people.cs.kuleuven.be/~tom.schrijvers/>
tom.schrijvers@cs.kuleuven.be

Abstract

Polymorphism and subtyping are important features in mainstream OO languages. The most common way to integrate the two is via $F_{<}$: style *bounded quantification*. A closely related mechanism is row polymorphism, which provides an alternative to subtyping, while still enabling many of the same applications. Yet another approach is to have type systems with *intersection types* and polymorphism. A recent addition to this design space are calculi with *disjoint intersection types* and *disjoint polymorphism*. With all these alternatives it is natural to wonder how they are related.

This paper provides *an* answer to this question. We show that disjoint polymorphism can recover forms of both row polymorphism and bounded polymorphism, while retaining key desirable properties, such as type-safety and decidability. Furthermore, we identify the extra power of disjoint polymorphism which enables additional features that cannot be easily encoded in calculi with row polymorphism or bounded quantification alone. Ultimately we expect that our work is useful to inform language designers about the expressive power of those common features, and to simplify implementations and metatheory of feature-rich languages with polymorphism and subtyping.

2012 ACM Subject Classification Theory of computation → Type theory; Software and its engineering → Object oriented languages; Software and its engineering → Polymorphism

Keywords and phrases Intersection types, bounded polymorphism, row polymorphism

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2020.27

Supplementary Material <https://github.com/xnning/Row-and-Bounded-via-Disjoint>

Funding This work has been sponsored by Hong Kong Research Grant Council projects number 17210617 and 17209519, and by the Research Foundation - Flanders.

Acknowledgements We thank the anonymous reviewers for their helpful comments.

1 Introduction

Intersection types [51, 22, 59] and *parametric polymorphism* are common features in many newer mainstream Object-Oriented (OO) languages. Among others intersection types are useful to express *multiple interface inheritance* [21]. They feature in programming languages like Scala [44], TypeScript [40], Ceylon [52] and Flow [31]. These languages also incorporate a form of *parametric polymorphism*, typically generalized to account for subtyping and supporting *bounded quantification* [12]. As programmers get more experienced with the



© Ningning Xie, Bruno C. d. S. Oliveira, Xuan Bi, and Tom Schrijvers;
licensed under Creative Commons License CC-BY

34th European Conference on Object-Oriented Programming (ECOOP 2020).

Editors: Robert Hirschfeld and Tobias Pape; Article No. 27; pp. 27:1–27:30

Leibniz International Proceedings in Informatics



LIPICs Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

27:2 Row and Bounded Polymorphism via Disjoint Polymorphism

combination of intersection types and polymorphism, they discover new applications. For example, the documentation of TypeScript [41] shows how the two features can express a composition operator for objects that enables an expressive form of statically typed *dynamic inheritance* [20, 32] and *mixin composition* [8]:

```
function extend<A, B>(first: A, second: B): A & B
```

The *polymorphic* function `extend` takes two objects and produces a result whose type is the intersection of the types of the original objects. The implementation of `extend` relies on low level features of JavaScript and is right-biased: the fields or properties of `second` are chosen in favor of the ones in `first`. For example, we can create a new object `jim` as follows:

```
var jim = extend(new Person('Jim'), new ConsoleLogger());
```

The `jim` object has type `Person & ConsoleLogger`, and acts both as a person and as a console logger. Using `extend` to compose objects is much more flexible than the *static inheritance* mechanisms of common OO languages like Java or Scala. It can type-check flexible OO patterns that have been used for many years in many dynamically-typed languages. Functions similar to `extend` have also been encoded in Scala [47, 54].

Unfortunately, the `extend` function in TypeScript suffers from *ambiguity* issues, and worse, it is not type-safe [2]. Indeed, given two objects with the same field or method names, `extend` does not detect potential conflicts. Instead it silently composes the two objects, using the implementation based on a biased choice. This does implement a mixin semantics, but it has the drawback that it can unintentionally override methods, without any warnings or errors. Additionally, the `extend` function is *not type-safe*: if two objects have the same property name with different types, `extend` may lookup the property of the wrong type.

In the literature of intersection types, `extend` is essentially what has been identified as the *merge operator* [55]. As illustrated by Dunfield [28], the expressive power of the merge operator can encode diverse programming language features, promising an economy of theory and implementation. Calculi with *disjoint intersection types* [46, 7, 2] incorporate a *coherent* merge operator. In such calculi the merge operator can merge two terms with *arbitrary* types as long as their types are disjoint; disjointness conflicts are reported as type-errors. Some calculi with disjoint intersection types, such as F_i^+ [7], also support *disjoint polymorphism* [2], which extends System F style universal quantification with a *disjointness constraint*. With disjoint polymorphism we can model `extend` as:

```
let extend A (B * A) (first : A, second : B) : A & B = first ,, second
```

Unlike the TypeScript definition, which relies on type-unsafe features, the definition above includes the full implementation. The definition of `extend` uses the merge operator `(,,)` to compose the two objects. The type variable `B` has a disjointness constraint `(B * A)` which states that `B` must be disjoint from `A`. Disjointness retains the flexibility to encode highly dynamic forms of inheritance, while ensuring both type-safety and the absence of conflicts.

Row polymorphism and disjoint polymorphism. Disjoint polymorphism looks quite close to certain forms of *row polymorphism*. Indeed, when restricted to *record types*, row polymorphism with *constrained quantification* [34] provides an approach to recovering an unambiguous semantics for `extend` as well. Constrained quantification extends System F style universal quantification with a *compatibility* constraint. By requiring `B` to be *compatible* with `A`, we can encode a row polymorphic variant of `extend` as:

```
let extend A (B # A) (first : A, second : B) : A || B = first || second
```

Here A and B are *row variables* standing for *record types*, and B is compatible with A ($B \# A$), which ensures the absence of conflicts. The $||$ operator concatenates two records at both the term level and the type level. The key difference between the two implementations of `extend` is that in the version with row variables, A and B only stand for record types. In contrast in the version with disjoint polymorphism, A and B are arbitrary types. In languages with nominal type systems, allowing arbitrary types is important to deal with nominal types of classes, for instance. The encoding of `extend` suggests that at least some functionality of row polymorphism can be captured with disjoint polymorphism. Indeed, there are clear analogies between the two mechanisms: the merge operators ($,$ and $||$) are similar; *compatibility* plays a similar role to *disjointness*; and intersection types generalize record type concatenation.

Bounded quantification and disjoint polymorphism. Polymorphic object-oriented languages also typically feature *bounded quantification*, which addresses the interaction between polymorphism and subtyping. Bounded quantification generalizes universal quantification by allowing programmers to specify upper bounds on type variables. For example:

```
let getName (A <: Person) (o : A) : (String,A) = (o.name,o)
```

expresses a function `getName` that takes an object o whose type is a subtype of `Person`, extracts its name and returns a copy of the object. Note that bounded quantification is useful to avoid the *loss of information problem* of subtyping [11]. Using the simpler type:

```
let getName_bad (o : Person) : (String,Person) = (o.name,o)
```

would lose static type information when given a *subtype* of `Person` as an argument.

An alternative version of `getName` that also does not lose type information is:

```
let getName A (o : A & Person) : (String,A & Person) = (o.name,o)
```

Here, the type variable A is unrestricted and represents the statically unknown part of the type of the object. The intersection type $A \& \text{Person}$ ensures that the object must at least contain all properties of `Person`, but does not forget about the statically unknown components. The two versions of `getName` show a common use case in OOP, but they use different features: the first uses *bounded quantification*, while the second uses a combination of intersection types and polymorphism. The connection between bounded quantification and polymorphic intersection types has been informally observed by Pierce [48].

Disjoint polymorphism, *row polymorphism* and *bounded quantification* provide a range of functionalities for OOP languages. Thus a language designer may be tempted to design a core language that combines all of these concepts. However, supporting all of them would lead to a significant implementation effort and a complex metatheory with non-trivial interactions between features. Furthermore, a common principle for (core) languages is to avoid overlapping features, which provide different ways to solve the same problem. Yet there seems to be a significant overlap between these features, which goes against that principle.

This paper builds on the similarities between the mechanisms, and shows that forms of both row polymorphism and bounded polymorphism can be recovered by type-safe elaborations into languages with disjoint polymorphism. Theoretically, it is important to formally establish the comparison among different type features, to allow a deep understanding and a precise discussion of the relative expressiveness of each feature. In practice, this result suggests

that core languages wishing to support all those features only need to support disjoint polymorphism natively, promising an economy of the implementation of those languages. To establish the relationship between row, bounded and disjoint polymorphism in a rigorous and precise manner, we formalize elaborations from λ^{\parallel} [34], a System F like calculus with row polymorphism, and from kernel $F_{<}$: [12], into F_i^+ . Our work serves as a guideline for language designers wishing to combine disjoint polymorphism, with bounded quantification and/or row polymorphism. The elaborations are useful to understand exactly what can and cannot be encoded, and to uncover and overcome difficulties. To our surprise, a full encoding of λ^{\parallel} is quite subtle: there are subtle differences between compatibility and disjointness. Moreover, certain general forms of bounded quantification are problematic, but all programs in kernel $F_{<}$: (the most widely used and decidable fragment of $F_{<}$.) are encodable.

We make the following specific contributions:

- **A formal elaboration from row to disjoint polymorphism:** We present a formal elaboration from λ^{\parallel} to F_i^+ (Section 4). We first identify an intuitive elaboration (Section 4.3). Due to discrepancies between compatibility and disjointness this elaboration does not work for all λ^{\parallel} programs. However it is possible to find a simple restriction on λ^{\parallel} that allows for the intuitive elaboration to work. We then present a complete, but *non-trivial* elaboration that targets the original λ^{\parallel} without restrictions (Section 4.4). While the design space of row polymorphic calculi is very diverse, features in λ^{\parallel} are representative of most other calculi. We discuss elaborating other row calculi in Section 6.1.
- **A formal elaboration from bounded to disjoint polymorphism:** We identify a fragment of $F_{<}$: that is encodable in terms of polymorphic intersection type systems, by providing an elaboration from *kernel* $F_{<}$: to F_i^+ (Section 5). Our elaboration, for the first time, validates the informal observation between polymorphic intersection systems and bounded quantification. We discuss other variants of $F_{<}$: in Section 6.2.
- **A discussion of the extra expressive power of disjoint polymorphism:** We identify and discuss specific features of disjoint polymorphism that cannot be easily encoded in $F_{<}$: and λ^{\parallel} (Section 2.4), including distributivity of intersections over other constructs, and the combination of subtyping and row polymorphism. We discuss other variants of intersection type systems in Section 6.3.
- **Coq formalization:** All elaborations and metatheory of this paper, except for some manual proof for simulation, has been mechanically formalized in the Coq proof assistant, including *type-safety* and *coherence*. The Coq formalization amounts to 18,855 lines of proofs and code (not including blank lines, comments and existing metatheory for F_i^+).

2 Overview

This section introduces the key ideas of the encodings for bounded quantification and row polymorphism. We also discuss the added extra power of disjoint polymorphism over bounded quantification and row polymorphism.

2.1 Background: Disjoint Polymorphism

Disjoint polymorphism [2, 7] combines disjoint intersection types with parametric polymorphism. In particular, F_i^+ [7] supports *intersection types* $A \& B$ for terms that are both of type A and of type B . With the *merge operator* we can construct terms of an intersection type, like $1, , \text{True}$ of type $\text{Int} \& \text{Bool}$. Thanks to *subtyping*, a term of type $\text{Int} \& \text{Bool}$ can also be used as if it had type Int , or as if it had type Bool . F_i^+ requires the two components of a merge to have disjoint types, e.g., $1, , 2 : \text{Int} \& \text{Int}$ is not allowed, because it is ambiguous which value

should be used at type `Int`. With *disjoint quantification*, it is possible to merge components whose type contains type variables. For instance, the term $\Lambda(\alpha * \text{Int}). \lambda(x : \alpha). x, , 1$ has type $\forall(\alpha * \text{Int}). \alpha \rightarrow \alpha \& \text{Int}$. The disjointness annotation $\alpha * \text{Int}$ allows α to be instantiated only to types that are disjoint from `Int`. Without a disjointness constraint, the term $\Lambda\alpha. \lambda(x : \alpha). x, , 1$ is rejected. Otherwise such a term would allow α to be instantiated to `Int`, and thus the function could be applied to numbers, e.g., `2`, leading to the *ambiguous* merge `2, , 1`.

2.2 Row Polymorphism through Disjoint Polymorphism

Row types, originally introduced by Wand [63] to model inheritance, provide an approach to typing extensible records. Row types have been studied extensively [35, 11, 53, 42] and have been applied to provide extensibility in various type systems [37, 36, 38]. According to Rémy [53], record calculi can be divided into those that support *free* extension, and those that support *strict* extension. The former allows extension with fields that already exist, whereas the latter does not. In this paper we focus on λ^{\parallel} , a calculus proposed by Harper and Pierce [34] that extends System F with row polymorphism. λ^{\parallel} belongs to the strict camp and avoids concatenating records with a field label in common by means of *constrained quantification*. A constrained quantifier attaches a constraint list to a type variable, which restricts the instantiations of that type variable to be record types with field labels that are distinct from all the record types in the constraint list. What sets λ^{\parallel} apart from other strict record calculi is its ability to merge records with statically unknown fields, and a mechanism to ensure that the resulting record is conflict-free (i.e., no duplicate labels). The following function concatenates two records by the *merge* operator \parallel :

$$\text{mergeRcd} = \Lambda(\alpha_1 \# \text{Empty}). \Lambda(\alpha_2 \# \alpha_1). \lambda(x_1 : \alpha_1). \lambda(x_2 : \alpha_2). x_1 \parallel x_2$$

which takes two type variables, each of which *lacks* ($\#$) the appropriate fields (`Empty` means no constraints at all). The function above can take any record type as its first argument, but the second type must be *compatible* with the first ($\alpha_2 \# \alpha_1$), i.e., the second record cannot have any labels that also occur in the first. These constraints ensure that the resulting record $x_1 \parallel x_2$ has no duplicate labels. If later we want to say that the first record x_1 has *at least* a field l_1 of type `Int`, we can refine the constraint list of α_1 , α_2 and the type of x_1 accordingly:

$$\Lambda(\alpha_1 \# \{l_1 : \text{Int}\}). \Lambda(\alpha_2 \# (\alpha_1, \{l_1 : \text{Int}\})). \lambda(x_1 : \alpha_1 \parallel \{l_1 : \text{Int}\}). \lambda(x_2 : \alpha_2). x_1 \parallel x_2$$

Encoding with disjoint polymorphism. Our encoding of λ^{\parallel} into F_i^+ is based on the similarities between the two calculi that the astute reader may have already observed. Indeed, the constrained quantification of record type variables $\Lambda(\alpha \# R). \varepsilon$ is quite similar to the disjoint quantification $\Lambda(\alpha * A). E$. They both constrain the use of respectively the record concatenation operator $x_1 \parallel x_2$ and the merge operator $x_1, , x_2$. Exploiting these similarities, we can encode `mergeRcd` as follows in F_i^+ :

$$\text{mergeAny} = \Lambda(\alpha_1 * \top). \Lambda(\alpha_2 * \alpha_1). \lambda(x_1 : \alpha_1). \lambda(x_2 : \alpha_2). x_1, , x_2$$

An important difference is that in `mergeRcd`, α_1 and α_2 are *row variables*: they can only be instantiated with record types. In contrast in `mergeAny`, α_1 and α_2 are type variables and they can be instantiated with any types, including types which are not records (such as `Int`).

Formal elaboration. To establish the validity of the encoding, we have formalized two different elaborations of λ^{\parallel} into F_i^+ . The first elaboration exploits the obvious similarity between the two mechanisms. While it clearly works for many example programs, the

formalization of the metatheory reveals that the straightforward elaboration does not work for all programs. Indeed, it turns out that there is a subtle difference in the interpretation of the constrained quantification and the disjoint quantification that makes the elaboration break down in some cases. For instance, the λ^{\parallel} binder $\Lambda\alpha\#\{l : \text{Int}\}$ expresses that α cannot have the label l *at all*. In contrast, the F_i^+ binder $\Lambda\beta * \{l : \text{Int}\}$ expresses that β cannot have a field l of type Int , but it can have a field l of some other *disjoint* type, say Bool . In what we consider to be contrived programs, this subtle difference invalidates the elaboration. We can eliminate this source of semantic difference by slightly restricting λ^{\parallel} , which is what we do in the first elaboration. However, in order to handle those contrived (but well-typed) unrestricted λ^{\parallel} programs as well, we also present a more complex elaboration that faithfully captures the semantics of constrained quantification in unrestricted λ^{\parallel} .

2.3 Bounded Quantification through Disjoint Polymorphism

Bounded quantification is a language feature that integrates parametric polymorphism with subtyping. It was first introduced in the language Fun [12] as a means of typing functions that operate uniformly over all subtypes of a given type, and has been the subject of much theoretical and practical effort [9, 48, 49, 39, 13, 11, 18, 25, 50]. In this paper, we focus on System $F_{<}$, which is a calculus with bounded quantification that extends System F.

As an illustration of bounded quantification, consider the following definition:

$$f = \lambda(x : \{\text{val} : \text{Int}\}). \{\text{orig} = x, \text{val} = x.\text{val} + 1\}$$

The function f has type $\{\text{val} : \text{Int}\} \rightarrow \{\text{orig} : \{\text{val} : \text{Int}\}, \text{val} : \text{Int}\}$, but it actually works for all records that have a val field of type Int . Thanks to bounded quantification we can formulate a variant of f that admits this:

$$fpoly = \Lambda(\alpha <: \{\text{val} : \text{Int}\}). \lambda(x : \alpha). \{\text{orig} = x, \text{val} = x.\text{val} + 1\}$$

The term $fpoly$ has type $\forall(\alpha <: \{\text{val} : \text{Int}\}). \alpha \rightarrow \{\text{orig} : \alpha, \text{val} : \text{Int}\}$. Here the (upper-)bound $\{\text{val} : \text{Int}\}$ restricts the instantiation of the quantified type variable α to subtypes of $\{\text{val} : \text{Int}\}$.

Encoding with disjoint polymorphism. Pierce [48] informally discussed an encoding of bounded quantification in terms of intersection types. To illustrate the encoding, let us consider a function of type $\forall(\alpha <: \text{Int}). \alpha \rightarrow \alpha$, which requires the type of the argument to be a subtype of Int . With intersection types, we know that $\alpha \& \text{Int}$ is always a subtype of Int . Therefore, the type $\forall\alpha. (\alpha \& \text{Int}) \rightarrow (\alpha \& \text{Int})$ expresses a similar subtype requirement. This leads to the following encoding of bounded quantification, by reading a bounded quantifier as an abbreviation for an unbounded one with a slightly modified body:

$$\forall(\alpha <: A). B \triangleq \forall\beta. ([\beta \& A/\alpha]B)$$

For the $fpoly$ example, we have its encoded type

$$\forall\beta. \beta \& \{\text{val} : \text{Int}\} \rightarrow \{\text{orig} : \beta \& \{\text{val} : \text{Int}\}, \text{val} : \text{Int}\}$$

However, there is no formalization of this encoding, and it is not clear at all what fragment of programs can be encoded. Pierce showed that this is not an encoding for full $F_{<}$: as it does not respect the subtyping rule for universal quantification. Nevertheless, after some experimentation, where the encoding was *manually* applied to complex examples, he came to the conclusion that “*the encoding trick works better than might be expected*”. Castagna and

Xu [19] even claim that “*bounded quantification does not require any modification*” in their intersection type system due to this encoding. However, due to Pierce’s counterexamples, without further qualification, this statement cannot be fully justified.

What is missing is to clarify precisely the expressiveness of this encoding with a type-theoretic formalization. Our work serves as a basis to fill the gaps, by identifying an encodable fragment of $F_{<}$, i.e., kernel $F_{<}$, and thus, for the first time, validates the informal observation of this encoding.

Formal elaboration. We formalize Pierce’s informal encoding idea and turn it into a structurally recursive procedure that systematically and simultaneously replaces all bounded quantifiers in a term. While doing this we faced several technical challenges. The first one was the misalignment between the $F_{<}$ and F_i^+ type systems: the former is undirected and the latter is bidirectional. This is a source of complication. In particular, we need to add explicit type annotations for all terms whose type cannot be synthesized, but only checked. Another challenge was the implicit use of subsumption in the typing of $F_{<}$ terms. We shift around the position in the term where subsumption happens and still arrive at the same type for the whole term. While the different typing derivations may lead to different F_i^+ elaborations, we do not want those different elaborations to have a different meaning. Hence, we must show that the elaboration is *coherent*. Finally we had to identify the class of $F_{<}$ programs for which the encoding actually works. This was not clear from the individual examples that Pierce gave, but it was necessary to make a formal statement that characterizes the extent and thus the usefulness of the encoding. Our translation shows that all well-typed kernel $F_{<}$ programs are encodable as well-typed F_i^+ programs. We believe that this justifies Pierce’s claim that the encoding might work better than expected, as kernel $F_{<}$ is the most common decidable fragment of $F_{<}$ and widely used to model key aspects of OO programs.

2.4 The Extra Power of Disjoint Polymorphism

This section identifies some of the additional expressive power of F_i^+ over $F_{<}$ and λ^{\parallel} alone.

Distributivity, Nested Composition and Family Polymorphism. F_i^+ is based on BCD subtyping [4], which features *distributive* subtyping rules, and enables *nested composition* of merges. Nested composition has several applications. In particular it is a key feature to enable *family polymorphism* [29].

With nested composition we can model a combinator that is useful to compose interpretations of *embedded DSLs*. A minimal example [7] is:

```
type R[e] = {lit : Int → e, neg : e → e} -- literal and negative expressions
compose =  $\Lambda(a * \top). \Lambda(b * a). \lambda(r1 : R[a]). \lambda(r2 : R[b]). (r1 \text{ ,, } r2) : R[a \& b]$ 
```

Here $R[e]$ stands for the abstract syntax of a tiny form of arithmetic expressions. The combinator `compose` allows the composition of two arbitrary interpretations (such as evaluation and pretty printing), into a single interpretation that runs both interpretations at once. In F_i^+ this functionality is achieved by simply merging `r1` and `r2`. Nested composition takes care of the details, by implicitly using a form of type-directed code generation, which is triggered by the upcast: $R[a] \& R[b] <: R[a \& b]$ in expression `r1 ,, r2`. The type of `r1 ,, r2` is $R[a] \& R[b]$. In F_i^+ , due to the distributivity properties of intersections, such a type is a subtype of $R[a \& b]$. Importantly, the fact that records are not treated specially in the type language is a key to allowing distributivity, which in turn enables nested composition.

27:8 Row and Bounded Polymorphism via Disjoint Polymorphism

The interested reader can see the work by Bi et al. [6, 7] for more complete examples. These examples illustrate how nested composition provides a simple and elegant solution to the *Expression Problem* (EP) [62]. In essence the approach mimics Ernst’s solution to the EP with family polymorphism [30] (which also relies on a form of nested composition).

With bounded quantification alone, `compose` is essentially not expressible. A solution with row polymorphism can be simulated only at the cost of more work:

```

Λ(a # Empty). Λ(b # a). λ(r1 : R[a]). λ(r2 : R[b]).
  { lit = λ(i : Int) . (r1.lit i      , r2.lit i)
    , neg = λ(e : (a, b)). (r1.neg (fst e), r2.neg (snd e)) }

```

Since row polymorphism does not support nested composition of merges, the code for executing the two interpretations at once has to be explicitly modeled with some tedious boilerplate code. Moreover, the results of the two interpretations have to be stored in a pair, and explicit projections are necessary to access the values.

In essence the manual composition approach employed with row polymorphism is akin to some existing solutions to the EP which need to tediously compose classes in different families manually. For instance, it is well-known that Scala enables solutions to the EP [65]. However, without nested composition those solutions are cluttered with manual composition code. In contrast, solutions based on nested composition are much more concise and elegant thanks to the automatic composition [30, 6, 7].

Subtyping and row typing. F_i^+ combines both subtyping and row polymorphism under one roof. The majority of systems with row polymorphism have been employed as an alternative to subtyping (although some row calculi also have subtyping, e.g., [11]). λ^{\parallel} , in particular, has no subtyping. One argument for row polymorphism is that it also eliminates the *loss of information problem* of subtyping [11]. For example, with subtyping, an identity function:

$$\lambda(x : \{l : \text{Int}\}). x$$

with type $\{l : \text{Int}\} \rightarrow \{l : \text{Int}\}$ may, inadvertently, lose some precision on the output type. For instance, the function can be applied to the record $\{l = 1, l' = \text{True}\}$, but the result type of such an application is $\{l : \text{Int}\}$ and not $\{l : \text{Int}, l' : \text{Bool}\}$.

λ^{\parallel} solves the loss of information problem by formulating the function in a different way:

$$\Lambda(\alpha \# \{l : \text{Int}\}). \lambda(x : \{l : \text{Int}\} \parallel \alpha). x$$

In this function the row variable α stands for any record without a label l . The type of x expresses that x includes a label l , as well as any labels in α . In this function the output type is $\{l : \text{Int}\} \parallel \alpha$ as well. Therefore the application of the function to $\{l = 1, l' = \text{True}\}$ has the type $\{l : \text{Int}, l' : \text{Bool}\}$, which does not lose precision.

In F_i^+ we can easily translate the λ^{\parallel} approach and reap its benefits too:

$$\Lambda(\alpha * \{l : \text{Int}\}). \lambda(x : \{l : \text{Int}\} \& \alpha). x$$

This function, like the row polymorphic version, preserves the precision of the output type.

Nevertheless, for many functions subtyping does not lose precision. For example:

$$\lambda(x : \{l : \text{Int}\}). x.l + 1$$

The function has type $\{l : \text{Int}\} \rightarrow \text{Int}$. In this case no matter which record is passed as an argument the output type is as precise as it can be. Note that this function is valid in F_i^+ and, because of subtyping, the record $\{l = 1, l' = \text{True}\}$ is a valid argument. However in λ^{\parallel} , the only way to allow records with more labels, is to generalize the function to:

$$\Lambda(\alpha \# \{l : \text{Int}\}). \lambda(x : \{l : \text{Int}\} \parallel \alpha). x.l + 1$$

In this case the generalization does not gain any precision, and in fact it requires a more complex type than the version with subtyping.

In summary, unlike λ^{\parallel} , many functions in F_i^+ can have a simpler non-polymorphic type and still allow for larger records to be used as inputs.

3 Disjoint Polymorphism

This section reviews F_i^+ , which serves as target of our elaborations of row and bounded polymorphism. The F_i^+ calculus and its metatheory have been studied already in Bi et al. [7]. We refer to prior work on for further details regarding F_i^+ 's formalization and metatheory.

3.1 Syntax and Semantics

Syntax. The syntax of F_i^+ is given at the top of Figure 1. Types A, B, C include integers Int , the top type \top , the bottom type \perp , arrows $A \rightarrow B$, intersection types $A \& B$, singleton record types $\{l : A\}$, type variables α and disjoint quantification $\forall(\alpha * A). B$. Expressions E include term variables x , integers i , the top value \top , abstractions $\lambda x. E$, applications $E_1 E_2$, merge expressions $E_1 , , E_2$, annotated terms $E : A$, singleton records $\{l = E\}$, record projections $E.l$, type abstractions $\Lambda(\alpha * A). E$ and type applications $E A$. Term contexts Γ record types of term variables, and type contexts Δ record disjointness constraints of type variables. Well-formedness of a type or a context are standard and omitted here.

Subtyping. The subtyping relation of F_i^+ is presented in the middle of Figure 1. Most rules are standard. For functions (rule S-ARR) and disjoint quantifications (rule S-FORALL), subtyping is covariant in positive positions, and contravariant in negative positions. Rules S-ANDL, S-ANDR, and S-AND for intersection types axiomatize that $A \& B$ is the greatest lower bound of A and B . Moreover, F_i^+ features BCD-style subtyping [4], where intersections are distributive over other type constructs. Concretely, intersections distribute over arrows (rule S-DISTARR), records (rule S-DISTRCD) and disjoint quantifications (rule S-DISTALL). Rules S-TOPARR, S-TOPRCD, and S-TOPALL are special cases of the distributivity rules, when viewing \top as a 0-ary intersection.

Typing. The bidirectional typing rules for F_i^+ are given at the bottom of Figure 1. The inference judgment $\Delta; \Gamma \vdash E \Rightarrow A$ says that under the type context Δ and the term context Γ , we can synthesize the type A for the expression E . The checking judgment $\Delta; \Gamma \vdash E \Leftarrow A$ checks E against the type A under the contexts Δ and Γ . Most of the typing rules are standard. Rule T-MERGE says that the merge expression $E_1 , , E_2$ is well-typed if both sub-expressions are well-typed, and their types are *disjoint*. The disjointness judgment $\Delta \vdash A_1 * A_2$ is important to rule out invalid merges, such as $1 , , 2$. Rule T-TABS says that, when typing a type abstraction, we put the disjointness constraint into the type context and then type-check the body. Conversely, rule T-TAPP checks that the type argument should satisfy the disjointness constraint.

Disjointness. Figure 2 presents the rules of the disjointness relation. Essentially, disjointness checks whether the merge of two expressions preserves coherence. Rules D-TOPL and D-TOPR say that *top-like* types are disjoint with any type. The top-like predicate $\lceil A \rceil$, given at

27:10 Row and Bounded Polymorphism via Disjoint Polymorphism

Types	$A, B, C ::= \text{Int} \mid \top \mid \perp \mid A \rightarrow B \mid A \& B \mid \{l : A\} \mid \alpha \mid \forall(\alpha * A). B$
Expressions	$E ::= x \mid i \mid \top \mid \lambda x. E \mid E_1 E_2 \mid E_1 ,, E_2 \mid E : A \mid \{l = E\} \mid E.l$ $\mid \Lambda(\alpha * A). E \mid EA$
Term contexts	$\Gamma ::= \bullet \mid \Gamma, x : A$
Type contexts	$\Delta ::= \bullet \mid \Delta, \alpha * A$

$A <: B$
(Declarative subtyping)

S-REFL $\frac{}{A <: A}$	S-TRANS $\frac{A_2 <: A_3 \quad A_1 <: A_2}{A_1 <: A_3}$	S-TOP $\frac{}{A <: \top}$	S-BOT $\frac{}{\perp <: A}$	S-RCD $\frac{}{\{l : A\} <: \{l : B\}}$
S-ARR $\frac{B_1 <: A_1 \quad A_2 <: B_2}{A_1 \rightarrow A_2 <: B_1 \rightarrow B_2}$	S-FORALL $\frac{B_1 <: B_2 \quad A_2 <: A_1}{\forall(\alpha * A_1). B_1 <: \forall(\alpha * A_2). B_2}$	S-AND $\frac{A_1 <: A_2 \quad A_1 <: A_3}{A_1 <: A_2 \& A_3}$		
S-ANDL $\frac{}{A_1 \& A_2 <: A_1}$	S-ANDR $\frac{}{A_1 \& A_2 <: A_2}$	S-DISTARR $\frac{}{(A_1 \rightarrow A_2) \& (A_1 \rightarrow A_3) <: A_1 \rightarrow A_2 \& A_3}$		
S-DISTRCD $\frac{}{\{l : A\} \& \{l : B\} <: \{l : A \& B\}}$		S-DISTALL $\frac{}{(\forall(\alpha * A). B_1) \& (\forall(\alpha * A). B_2) <: \forall(\alpha * A). B_1 \& B_2}$		
S-TOPARR $\frac{}{\top <: \top \rightarrow \top}$		S-TOPRCD $\frac{}{\top <: \{l : \top\}}$	S-TOPALL $\frac{}{\top <: \forall(\alpha * \top). \top}$	

$\Delta; \Gamma \vdash E \Rightarrow A$
(Inference)

T-TOP $\frac{\vdash \Delta \quad \Delta \vdash \Gamma}{\Delta; \Gamma \vdash \top \Rightarrow \top}$	T-NAT $\frac{\vdash \Delta \quad \Delta \vdash \Gamma}{\Delta; \Gamma \vdash i \Rightarrow \text{Int}}$	T-VAR $\frac{\vdash \Delta \quad \Delta \vdash \Gamma \quad (x : A) \in \Gamma}{\Delta; \Gamma \vdash x \Rightarrow A}$
T-APP $\frac{\Delta; \Gamma \vdash E_1 \Rightarrow A_1 \rightarrow A_2 \quad \Delta; \Gamma \vdash E_2 \Leftarrow A_1}{\Delta; \Gamma \vdash E_1 E_2 \Rightarrow A_2}$	T-TABS $\frac{\Delta \vdash A \quad \Delta, \alpha * A; \Gamma \vdash E \Rightarrow B}{\Delta; \Gamma \vdash \Lambda(\alpha * A). E \Rightarrow \forall(\alpha * A). B}$	
T-MERGE $\frac{\Delta; \Gamma \vdash E_1 \Rightarrow A_1 \quad \Delta; \Gamma \vdash E_2 \Rightarrow A_2 \quad \Delta \vdash A_1 * A_2}{\Delta; \Gamma \vdash E_1 ,, E_2 \Rightarrow A_1 \& A_2}$		T-RCD $\frac{\Delta; \Gamma \vdash E \Rightarrow A}{\Delta; \Gamma \vdash \{l = E\} \Rightarrow \{l : A\}}$
T-PROJ $\frac{\Delta; \Gamma \vdash E \Rightarrow \{l : A\}}{\Delta; \Gamma \vdash E.l \Rightarrow A}$	T-ANNO $\frac{\Delta; \Gamma \vdash E \Leftarrow A}{\Delta; \Gamma \vdash E : A \Rightarrow A}$	T-TAPP $\frac{\Delta; \Gamma \vdash E \Rightarrow \forall(\alpha * B). C \quad \Delta \vdash A * B}{\Delta; \Gamma \vdash EA \Rightarrow [A/\alpha]C}$

$\Delta; \Gamma \vdash E \Leftarrow A$
(Checking)

T-ABS $\frac{\Delta \vdash A \quad \Delta; \Gamma, x : A \vdash E \Leftarrow B}{\Delta; \Gamma \vdash \lambda x. E \Leftarrow A \rightarrow B}$	T-SUB $\frac{\Delta; \Gamma \vdash E \Rightarrow B \quad B <: A}{\Delta; \Gamma \vdash E \Leftarrow A}$
--	--

■ **Figure 1** Syntax, declarative subtyping, and bidirectional type system of F_i^+ .

$$\boxed{\lceil A \rceil} \quad (Top\text{-}like\ types)$$

$$\begin{array}{c}
\text{TL-TOP} \\
\frac{}{\lceil \top \rceil}
\end{array}
\quad
\begin{array}{c}
\text{TL-AND} \\
\frac{\lceil A \rceil \quad \lceil B \rceil}{\lceil A \& B \rceil}
\end{array}
\quad
\begin{array}{c}
\text{TL-ARR} \\
\frac{\lceil B \rceil}{\lceil A \rightarrow B \rceil}
\end{array}
\quad
\begin{array}{c}
\text{TL-RCD} \\
\frac{\lceil A \rceil}{\lceil \{l : A\} \rceil}
\end{array}
\quad
\begin{array}{c}
\text{TL-ALL} \\
\frac{\lceil B \rceil}{\lceil \forall(\alpha * A). B \rceil}
\end{array}$$

$$\boxed{\Delta \vdash A * B} \quad (Disjointness)$$

$$\begin{array}{c}
\text{D-TOPL} \\
\frac{\lceil A \rceil}{\Delta \vdash A * B}
\end{array}
\quad
\begin{array}{c}
\text{D-TOPR} \\
\frac{\lceil B \rceil}{\Delta \vdash A * B}
\end{array}
\quad
\begin{array}{c}
\text{D-AX} \\
\frac{A *_{ax} B}{\Delta \vdash A * B}
\end{array}
\quad
\begin{array}{c}
\text{D-ARR} \\
\frac{\Delta \vdash A_2 * B_2}{\Delta \vdash A_1 \rightarrow A_2 * B_1 \rightarrow B_2}
\end{array}$$

$$\begin{array}{c}
\text{D-ANDL} \\
\frac{\Delta \vdash A_1 * B \quad \Delta \vdash A_2 * B}{\Delta \vdash A_1 \& A_2 * B}
\end{array}
\quad
\begin{array}{c}
\text{D-ANDR} \\
\frac{\Delta \vdash A * B_1 \quad \Delta \vdash A * B_2}{\Delta \vdash A * B_1 \& B_2}
\end{array}
\quad
\begin{array}{c}
\text{D-RCDNEQ} \\
\frac{l_1 \neq l_2}{\Delta \vdash \{l_1 : A\} * \{l_2 : B\}}
\end{array}$$

$$\begin{array}{c}
\text{D-RCD EQ} \\
\frac{\Delta \vdash A * B}{\Delta \vdash \{l : A\} * \{l : B\}}
\end{array}
\quad
\begin{array}{c}
\text{D-TVARL} \\
\frac{(\alpha * A) \in \Delta \quad A <: B}{\Delta \vdash \alpha * B}
\end{array}
\quad
\begin{array}{c}
\text{D-TVARR} \\
\frac{(\alpha * A) \in \Delta \quad A <: B}{\Delta \vdash B * \alpha}
\end{array}$$

$$\begin{array}{c}
\text{D-FORALL} \\
\frac{\Delta, \alpha * A_1 \& A_2 \vdash B_1 * B_2}{\Delta \vdash \forall(\alpha * A_1). B_1 * \forall(\alpha * A_2). B_2}
\end{array}$$

■ **Figure 2** Selected rules for disjointness.

the top of Figure 2, captures the set of types that are isomorphic to \top . Disjointness axioms $A *_{ax} B$ (appearing in rule D-AX) take care of two types with different type constructors (e.g., Int and records). The axiom rules can be found in Appendix A.2. The other disjointness rules are standard and explained in detail in previous work [46, 2]. Finally, we note that subtyping preserves disjointness.

► **Lemma 1** (Subtyping preserves disjointness). *If $\Delta \vdash A * B$ and $B <: C$, then $\Delta \vdash A * C$.*

3.2 Elaboration and Coherence

The dynamic semantics of F_i^+ is given by a type-directed elaboration ($\rightsquigarrow e$) into another calculus, F_{co} , a variant of System F with explicit coercions. The full definition of F_{co} and the elaboration process can be found in Appendix B. The main challenge of the elaboration is that, due to the non-deterministic nature of the declarative type system, an F_i^+ expression can elaborate to different F_{co} expressions. For example, the subtyping rules S-AND, S-ANDL, and S-ANDR overlap with each other when both sides are intersections, leading to different coercions depending on the order in which these rules are applied. To establish coherence for F_i^+ , Bi et al. [7] resort to contextual equivalence, and they prove that different elaborations of the same F_i^+ expression are contextually equivalent. More formally, $\Delta; \Gamma \vdash e_1 \simeq_{ctx} e_2$ means that two F_{co} expressions are contextually equivalent under the corresponding elaboration contexts of Δ and Γ . We state the central coherence theorem below.

► **Theorem 2** (Coherence of F_i^+). *We have that*

- If $\Delta; \Gamma \vdash E \Rightarrow A \rightsquigarrow e_1$, and $\Delta; \Gamma \vdash E \Rightarrow A \rightsquigarrow e_2$, then $\Delta; \Gamma \vdash e_1 \simeq_{ctx} e_2$.
- If $\Delta; \Gamma \vdash E \Leftarrow A \rightsquigarrow e_1$, and $\Delta; \Gamma \vdash E \Leftarrow A \rightsquigarrow e_2$, then $\Delta; \Gamma \vdash e_1 \simeq_{ctx} e_2$.

4 Encoding Row Polymorphism

This section shows how to systematically elaborate λ^{\parallel} [34] – a polymorphic record calculus with *constrained quantification* – into F_i^+ . We first identify a simple and direct elaboration for a fragment of λ^{\parallel} , and then present a carefully crafted elaboration of full λ^{\parallel} using a more sophisticated elaboration.

4.1 Syntax of λ^{\parallel}

We start by briefly reviewing the syntax of λ^{\parallel} , shown at the top of Figure 3. Metavariable t ranges over types, which include the integer type `Int`, function types $t_1 \rightarrow t_2$, constrained quantifications $\forall \alpha \# R. t$ and record types r . Record types are built from record type variables α , the empty record type `Empty`, single-field records $\{l : t\}$ and record merges $r_1 \parallel r_2$.¹ A constraint list R of record types is used to constrain instantiations of record type variables.

Metavariable ε ranges over terms, including term variables x , integers i , lambda abstractions $\lambda(x : t). \varepsilon$, function applications $\varepsilon_1 \varepsilon_2$, the empty record `empty`, single-field records $\{l = \varepsilon\}$, record merges $\varepsilon_1 \parallel \varepsilon_2$, record restrictions $\varepsilon \setminus l$, record projections $\varepsilon.l$, type abstractions $\Lambda(\alpha \# R). \varepsilon$ and type applications $\varepsilon [r]$. As a side note, from the syntax of type applications $\varepsilon [r]$, it can already be seen that λ^{\parallel} only supports quantification over *record types*.

4.2 Typing Rules of λ^{\parallel}

The type system of λ^{\parallel} consists of several conventional judgments. The complete set of rules appears in Appendix C.2. Figure 3 presents selected well-formedness rules for record types. A merge $r_1 \parallel r_2$ is well-formed in context T if r_1 and r_2 are well-formed, and moreover, r_1 and r_2 are compatible in T (rule WFR-MERGE) – the most important judgment in λ^{\parallel} , as we will explain next.

Compatibility. The compatibility relation in the middle of Figure 3 plays a central role in λ^{\parallel} . It is the underlying mechanism for deciding when merging two records is “sensible”. Informally, $T \vdash r_1 \# r_2$ holds if r_1 lacks every field contained in r_2 and vice versa. Compatibility is symmetric (rule CMP-SYMM) and respects type equivalence (rule CMP-EQ). Rule CMP-BASE says that if a record is compatible with $\{l : t\}$, it is also compatible with every record $\{l : t'\}$ with the same label l . A type variable is compatible with the records in its constraint list (rule CMP-TVAR). Two single-field records are compatible if they have different labels (rule CMP-BASEBASE). The remaining rules are self-explanatory; we refer the reader to [34] for further explanation. The judgment of constraint list satisfaction $T \vdash r \# R$ ensures that r is compatible with every record in the constraint list R .

Type equivalence. Unlike F_i^+ , λ^{\parallel} does not have subtyping. Instead, λ^{\parallel} uses type equivalence to convert terms of one type to another. A selection of the rules defining equivalence of types and constraint lists appears at the bottom of Figure 3. The relation $t_1 \sim t_2$ is an equivalence relation, and is a congruence with respect to the type constructors. Merge is associative (rule TEQ-MERGEASSOC), commutative (rule TEQ-MERGECOMM), and has `Empty` as its unit (rule TEQ-MERGEUNIT). As a consequence, records are identified up to permutations. The equivalence of constrained quantification (rule TEQ-CONGALL) relies on the equivalence of

¹ The original λ^{\parallel} also includes record type restrictions $r \setminus l$, which can be systematically erased using type equivalence, thus we omit type-level restrictions but keep term-level restrictions.

Types	$t ::= \text{Int} \mid t_1 \rightarrow t_2 \mid \forall \alpha \# R. t \mid r$
Records	$r ::= \alpha \mid \text{Empty} \mid \{l : t\} \mid r_1 \parallel r_2$
Constraint lists	$R ::= \diamond \mid r, R$
Terms	$\varepsilon ::= x \mid i \mid \lambda(x : t). \varepsilon \mid \varepsilon_1 \varepsilon_2 \mid \text{empty} \mid \{l = \varepsilon\} \mid \varepsilon_1 \parallel \varepsilon_2$ $\mid \varepsilon \setminus l \mid \varepsilon.l \mid \Lambda(\alpha \# R). \varepsilon \mid \varepsilon[r]$
Term contexts	$G ::= \diamond \mid G, x : t$
Type contexts	$T ::= \diamond \mid T, \alpha \# R$

 $T \vdash r \text{ record}$ *(Well-formed record types)*

$$\frac{\text{WFR-VAR} \quad (\alpha \# R) \in T}{T \vdash \alpha \text{ record}} \quad \frac{\text{WFR-MERGE} \quad T \vdash r_1 \text{ record} \quad T \vdash r_2 \text{ record} \quad T \vdash r_1 \# r_2}{T \vdash r_1 \parallel r_2 \text{ record}}$$

 $T \vdash r_1 \# r_2$ *(Compatibility)*

$$\frac{\text{CMP-EQ} \quad T \vdash r \# s \quad r \sim r' \quad s \sim s'}{T \vdash r' \# s'}$$

$$\frac{\text{CMP-SYMM} \quad T \vdash r \# s}{T \vdash s \# r}$$

$$\frac{\text{CMP-BASE} \quad T \vdash r \# \{l : t\} \quad T \vdash t' \text{ type}}{T \vdash r \# \{l : t'\}}$$

$$\frac{\text{CMP-TVAR} \quad (\alpha \# R) \in T \quad T \vdash R \text{ ok} \quad r \in R}{T \vdash \alpha \# r}$$

$$\frac{\text{CMP-MERGE} \quad T \vdash r \# (s_1 \parallel s_2)}{T \vdash r \# s_i}$$

$$\frac{\text{CMP-EMPTY} \quad T \vdash r \text{ record}}{T \vdash r \# \text{Empty}}$$

$$\frac{\text{CMP-MERGEI} \quad T \vdash s_1 \# s_2 \quad T \vdash r \# s_1 \quad T \vdash r \# s_2}{T \vdash r \# (s_1 \parallel s_2)}$$

$$\frac{\text{CMP-BASEBASE} \quad l \neq l' \quad T \vdash t \text{ type} \quad T \vdash t' \text{ type}}{T \vdash \{l : t\} \# \{l' : t'\}}$$

 $T \vdash r \# R$ *(Constraint list satisfaction)*

$$\frac{\text{CMLIST-NIL} \quad T \vdash r \text{ record}}{T \vdash r \# \diamond}$$

$$\frac{\text{CMLIST-CONS} \quad T \vdash r \# r' \quad T \vdash r \# R}{T \vdash r \# r', R}$$

 $t_1 \sim t_2$ *(Type equivalence)*

$$\frac{\text{TEQ-MERGEASSOC}}{r_1 \parallel (r_2 \parallel r_3) \sim (r_1 \parallel r_2) \parallel r_3}$$

$$\frac{\text{TEQ-MERGECOMM}}{r_1 \parallel r_2 \sim r_2 \parallel r_1}$$

$$\frac{\text{TEQ-MERGEUNIT}}{r \parallel \text{Empty} \sim r}$$

$$\frac{\text{TEQ-CONGALL} \quad R \sim R' \quad t \sim t'}{\forall \alpha \# R. t \sim \forall \alpha \# R'. t'}$$

 $R_1 \sim R_2$ *(Constraint list equivalence)*

$$\frac{\text{CEQ-SWAP}}{r, (r', R) \sim r', (r, R)}$$

$$\frac{\text{CEQ-MERGE}}{(r_1 \parallel r_2), R \sim r_1, (r_2, R)}$$

$$\frac{\text{CEQ-EMPTY}}{\text{Empty}, R \sim R}$$

$$\frac{\text{CEQ-BASE}}{\{l : t\}, R \sim \{l : t'\}, R}$$

■ **Figure 3** Syntax, and selected rules of λ^{\parallel} .

$$\boxed{T; G \vdash \varepsilon : t \rightsquigarrow E} \quad (\text{Type-directed elaboration})$$

$$\begin{array}{c}
\text{WTT-EQ} \\
\frac{T; G \vdash \varepsilon : t \rightsquigarrow E \quad T \vdash t' \text{ type} \quad t \sim t'}{T; G \vdash \varepsilon : t' \rightsquigarrow E : \llbracket t' \rrbracket}
\end{array}
\qquad
\begin{array}{c}
\text{WTT-BASE} \\
\frac{T; G \vdash \varepsilon : t \rightsquigarrow E}{T; G \vdash \{l = \varepsilon\} : \{l : t\} \rightsquigarrow \{l = E\}}
\end{array}$$

$$\begin{array}{c}
\text{WTT-RESTR} \\
\frac{T; G \vdash \varepsilon : \{l : t\} \parallel r \rightsquigarrow E}{T; G \vdash \varepsilon \setminus l : r \rightsquigarrow E : \llbracket r \rrbracket}
\end{array}
\qquad
\begin{array}{c}
\text{WTT-SELECT} \\
\frac{T; G \vdash \varepsilon : \{l : t\} \parallel r \rightsquigarrow E}{T; G \vdash \varepsilon.l : t \rightsquigarrow (E : \{l : \llbracket t \rrbracket\}).l}
\end{array}
\qquad
\begin{array}{c}
\text{WTT-EMPTY} \\
\frac{T \text{ ok} \quad T \vdash G \text{ ok}}{T; G \vdash \text{empty} : \text{Empty} \rightsquigarrow \top}
\end{array}$$

$$\begin{array}{c}
\text{WTT-MERGE} \\
\frac{T; G \vdash \varepsilon_1 : r_1 \rightsquigarrow E_1 \quad T \vdash r_1 \# r_2}{T; G \vdash \varepsilon_1 \parallel \varepsilon_2 : r_1 \parallel r_2 \rightsquigarrow E_1, E_2}
\end{array}
\qquad
\begin{array}{c}
\text{WTT-ALLE} \\
\frac{T; G \vdash \varepsilon : \forall \alpha \# R. t \rightsquigarrow E \quad T \vdash r \# R}{T; G \vdash \varepsilon [r] : [r/\alpha]t \rightsquigarrow E \llbracket r \rrbracket \llbracket r \rrbracket_{\perp}}
\end{array}$$

$$\begin{array}{c}
\text{WTT-ALLI} \\
\frac{T \vdash R \text{ ok} \quad T, \alpha \# R; G \vdash \varepsilon : t \rightsquigarrow E}{T; G \vdash \Lambda(\alpha \# R). \varepsilon : \forall \alpha \# R. t \rightsquigarrow \Lambda(\alpha * \llbracket R \rrbracket). \Lambda(\alpha_{\perp} * \llbracket R \rrbracket). E}
\end{array}$$

■ **Figure 4** Selected typing rules of λ^{\parallel} with elaboration.

constraint lists $R_1 \sim R_2$. Again, it is an equivalence relation, and it respects type equivalence. Constraint lists are essentially finite sets, so order is irrelevant (rule CEQ-SWAP). Merges of constraints can be “flattened” (rule CEQ-MERGE), and occurrences of **Empty** may be eliminated (rule CEQ-EMPTY). The last rule CEQ-BASE is quite interesting: it implies that the types of single-field records are ignored. The reason is that, as far as compatibility is concerned, only labels matter, thus changing the types of records in constraint lists will not affect their compatibility relation. We will have more to say about this in Section 4.3.

Typing rules. A selection of typing rules is shown in Figure 4. In a first reading, the gray parts can be ignored. Most of the typing rules are quite standard. Typing is invariant under type equivalence (rule WTT-EQ). Two terms can be merged if their types are compatible (rule WTT-MERGE). Type application $\varepsilon [r]$ is well-typed if the type argument r satisfies the constraints R (rule WTT-ALLE).

► **Remark 3.** We have made a few simplifications compared to the original λ^{\parallel} , notably the typing of record selection (rule WTT-SELECT) and restriction (rule WTT-RESTR). In the original formulation, both typing rules use a partial function r_l that denotes the type associated with label l in r . Instead of using partial functions, here we explicitly expose the expected label in a record. It can be shown that if label l is present in record type r , then the fields in r can be rearranged so that l comes first by type equivalence. This formulation was also adopted by Leijen [35].

4.3 A Simple yet Incomplete Encoding

The similarities between λ^{\parallel} and F_i^+ , which the astute reader may have already observed, suggest an intuitive elaboration scheme. On the syntactic level, it is easy to see a one-to-one correspondence between λ^{\parallel} types and F_i^+ types. We use $\llbracket t \rrbracket$ to denote the elaboration

$\boxed{\llbracket t \rrbracket} \quad \llbracket \text{Int} \rrbracket = \text{Int}$ $\llbracket t_1 \rightarrow t_2 \rrbracket = \llbracket t_1 \rrbracket \rightarrow \llbracket t_2 \rrbracket$ $\llbracket \forall \alpha \# R. t \rrbracket = \forall (\alpha * \llbracket R \rrbracket). \llbracket t \rrbracket$ $\llbracket \alpha \rrbracket = \alpha$ $\llbracket \text{Empty} \rrbracket = \top$ $\llbracket \{l : t\} \rrbracket = \{l : \llbracket t \rrbracket\}$ $\llbracket r_1 \parallel r_2 \rrbracket = \llbracket r_1 \rrbracket \& \llbracket r_2 \rrbracket$	$\boxed{\llbracket R \rrbracket} \quad \llbracket \diamond \rrbracket = \top$ $\llbracket r, R \rrbracket = \llbracket r \rrbracket \& \llbracket R \rrbracket$ $\boxed{\llbracket T \rrbracket} \quad \llbracket \diamond \rrbracket = \bullet$ $\llbracket T, \alpha \# R \rrbracket = \llbracket T \rrbracket, \alpha * \llbracket R \rrbracket$ $\boxed{\llbracket G \rrbracket} \quad \llbracket \diamond \rrbracket = \bullet$ $\llbracket G, x : t \rrbracket = \llbracket G \rrbracket, x : \llbracket t \rrbracket$
$\boxed{T; G \vdash \varepsilon : t \rightsquigarrow_i E}$	<i>(Type-directed elaboration)</i>
$\text{WTTI-EQ} \quad \frac{T; G \vdash \varepsilon : t \rightsquigarrow_i E \quad T \vdash t' \text{ type} \quad t \sim t'}{T; G \vdash \varepsilon : t' \rightsquigarrow_i E : \llbracket t' \rrbracket}$	$\text{WTTI-BASE} \quad \frac{T; G \vdash \varepsilon : t \rightsquigarrow_i E}{T; G \vdash \{l = \varepsilon\} : \{l : t\} \rightsquigarrow_i \{l = E\}}$
$\text{WTTI-ALLI} \quad \frac{T \vdash R \text{ ok} \quad T, \alpha \# R; G \vdash \varepsilon : t \rightsquigarrow_i E}{T; G \vdash \Lambda(\alpha \# R). \varepsilon : \forall \alpha \# R. t \rightsquigarrow_i \Lambda(\alpha * \llbracket R \rrbracket). E}$	$\text{WTTI-ALLE} \quad \frac{T; G \vdash \varepsilon : \forall \alpha \# R. t \rightsquigarrow_i E \quad T \vdash r \# R}{T; G \vdash \varepsilon [r] : [r/\alpha]t \rightsquigarrow_i E \llbracket r \rrbracket}$

■ **Figure 5** Intuitive elaboration functions, and selected type-directed elaboration from λ^{\parallel} to F_i^+ .

function from λ^{\parallel} types to F_i^+ types, whose formal definition is given at the top of Figure 5. Elaboration of expressions is also easy. Constrained type abstractions $\Lambda(\alpha \# R). \varepsilon$ correspond to $\Lambda(\alpha * A). E$; record merges can be simulated by the more general merge operator of F_i^+ ; record restriction can be modeled as annotated terms, and so on. On the semantic level, well-formedness judgments of λ^{\parallel} match with well-formedness judgments of F_i^+ . The compatibility relation corresponds to the disjointness relation. What might not be so obvious is that type equivalence is expressible via subtyping. More specifically, $t_1 \sim t_2$ induces two subtyping relations: $\llbracket t_1 \rrbracket <: \llbracket t_2 \rrbracket$ and $\llbracket t_2 \rrbracket <: \llbracket t_1 \rrbracket$. Under this elaboration scheme, the full definition of type-directed elaboration, denoted as $T; G \vdash \varepsilon : t \rightsquigarrow_i E$, where i stands for “intuitive”, is simple (selected rules are given at the bottom of Figure 5). With all these in mind, let us consider two examples.

► **Example 4.** Consider the term $\Lambda(\alpha \# \{l : \text{Int}\}). \lambda(x : \alpha). x$. This term can be assigned the type (among others) $\forall \alpha \# \{l : \text{Int}\}. \alpha \rightarrow \alpha$, and its F_i^+ counterpart $\Lambda(\alpha * \{l : \text{Int}\}). \lambda(x : \alpha). x$ has type $\forall (\alpha * \{l : \text{Int}\}). \alpha \rightarrow \alpha$, which corresponds directly to $\forall \alpha \# \{l : \text{Int}\}. \alpha \rightarrow \alpha$. In λ^{\parallel} , the same term could also be assigned type $\forall \alpha \# \{l : \text{Bool}\}. \alpha \rightarrow \alpha$ (rule WTT-EQ), since $\forall \alpha \# \{l : \text{Bool}\}. \alpha \rightarrow \alpha$ is equivalent to $\forall \alpha \# \{l : \text{Int}\}. \alpha \rightarrow \alpha$ by rules TEQ-CONGALL and CEQ-BASE. However, in F_i^+ , these two types have no relationship at all – $\forall (\alpha * \{l : \text{Int}\}). \alpha \rightarrow \alpha$ is not the same as $\forall (\alpha * \{l : \text{Bool}\}). \alpha \rightarrow \alpha$, and indeed it should not be, as these two types have completely different meanings!

► **Example 5.** Consider the term $\varepsilon = \Lambda(\alpha \# \{l : \text{Bool}\}). \lambda(x : \alpha). \lambda(y : \{l : \text{Int}\}). x \parallel y$. This term has type $\forall \alpha \# \{l : \text{Bool}\}. \alpha \rightarrow \{l : \text{Int}\} \rightarrow \alpha \parallel \{l : \text{Int}\}$, and its “obvious” elaboration is $E = \Lambda(\alpha * \{l : \text{Bool}\}). \lambda(x : \alpha). \lambda(y : \{l : \text{Int}\}). x, , y$. However, expression E is ill-typed in F_i^+ : we *cannot* merge x with y because their types (α and $\{l : \text{Int}\}$ respectively) are not disjoint. Allowing it to type-check causes incoherence: evaluating $(E \{l : \text{Int}\} \{l = 1\} \{l = 2\}). l$ could result in 1 or 2!

These examples underline a crucial observation: disjointness is more *fine-grained* than compatibility. Unlike F_i^+ , the existence of ε in λ^{\parallel} will not cause incoherence because compatibility can only relate records with different labels, and thus ε can only be applied to records without label l at all. So λ^{\parallel} rejects type application $\varepsilon[\{l : \text{Int}\}]$ in the first place. However, disjointness also relates records with the same label as long as their types are disjoint, i.e., rule D-RCDEQ. Section 2.4 illustrates the importance of rule D-RCDEQ for distributivity, which is not supported by λ^{\parallel} . A careful comparison between the two calculi reveals that two rules are “to blame”: rule CEQ-BASE and rule CMP-BASE, which are the cause for the problem in Example 4 and Example 5 respectively.

$$\frac{}{\{l : t\}, R \sim \{l : t'\}, R} \text{CEQ-BASE} \qquad \frac{T \vdash r \# \{l : t\} \quad T \vdash t' \text{ type}}{T \vdash r \# \{l : t'\}} \text{CMP-BASE}$$

Yet, both Example 4 and Example 5 seem contrived. From the expression $\Lambda(\alpha \# \{l : \text{Int}\}). \lambda(x : \alpha). x$, the user can reasonably expect the type to be $\forall \alpha \# \{l : \text{Int}\}. \alpha \rightarrow \alpha$. For ε , an *equivalent* definition with more sensible and readable annotation is $\varepsilon' = \Lambda(\alpha \# \{l : \text{Int}\}). \lambda(x : \alpha). \lambda(y : \{l : \text{Int}\}). x \parallel y$, whose corresponding elaboration type-checks successfully. We believe that programs with the same issue always have some *equivalent* accepted programs by changing some type annotations.

We propose a restricted λ^{\parallel} by: (1) replacing rule CEQ-BASE with rule CEQ-BASEALT; and (2) removing rule CMP-BASE. We conjecture that this change has no practical consequences and no expressiveness is lost. Moreover, the restrictions coincide with the observation in Harper and Pierce [34]: *we may normalize constraint lists into the form $l_1, \dots, l_n, \alpha_1, \dots, \alpha_m$ where the l_i 's are labels and the α_i 's are record type variables*. The normalization then validates the change of rules.

$$\frac{t \sim t'}{\{l : t\}, R \sim \{l : t'\}, R} \text{CEQ-BASEALT}$$

In return, we can prove the intuitive elaboration for restricted λ^{\parallel} is, indeed, sound:

► **Theorem 6** (Type-safety of \rightsquigarrow_i elaboration). *If $T; G \vdash \varepsilon : t \rightsquigarrow_i E$ then $\llbracket T \rrbracket; \llbracket G \rrbracket \vdash E \Rightarrow \llbracket t \rrbracket$.*

4.4 A Complete Encoding of λ^{\parallel} and its Challenges

One criticism to the intuitive encoding is that it does not fully model λ^{\parallel} : fewer expressions type-check in the modified λ^{\parallel} . Thus, we present a carefully designed encoding that is able to elaborate the original λ^{\parallel} to F_i^+ *without* any restrictions at all. It is highly non-trivial and reveals the essence of constrained quantification from the point of view of disjointness.

First, let us take a step back and have another look at Example 5. As we have discussed, the root cause is rule CMP-BASE, which says that *if a record is compatible with a single-field record $\{l : t\}$, then it is compatible with every single-field record $\{l : t'\}$* . To express the essence of rule CMP-BASE in F_i^+ , we utilize the bottom type \perp . In F_i^+ , according to Lemma 1, if some type A is disjoint to $\{l : \perp\}$, then, because $\{l : \perp\} < \{l : B\}$ (by rules S-RCD and S-BOT) for any B , we have that A is disjoint to $\{l : B\}$. In other words, in F_i^+ , *if a record is disjoint to $\{l : \perp\}$, then it is disjoint to every single-field record $\{l : A\}$* .

► **Lemma 7** (Disjointness to records with bottom). *If $\Delta \vdash A * \{l : \perp\}$, then $\Delta \vdash A * \{l : B\}$ for all B .*

Essentially, a compatibility constraint with $\{l : t\}$ in λ^{\parallel} corresponds to a disjointness constraint to $\{l : \perp\}$ in F_i^+ . Thus, we *bottom-elaborate* the record types that *appear in a constraint list*: if a record $\{l : t\}$ appears in a constraint list, then it is bottom-elaborated to $\{l : \perp\}$. For Example 4, both $\forall \alpha \# \{l : \text{Int}\}. \alpha \rightarrow \alpha$ and $\forall \alpha \# \{l : \text{Bool}\}. \alpha \rightarrow \alpha$ elaborate to $\forall (\alpha * \{l : \perp\}). \alpha \rightarrow \alpha$. For Example 5, ε elaborates to $E' = \Lambda(\alpha * \{l : \perp\}). \lambda(x : \alpha). \lambda(y : \{l : \text{Int}\}). x, y$, which type-checks in F_i^+ .

► **Example 8.** Now consider the λ^{\parallel} term

$$\varepsilon_1 = (\Lambda(\alpha \# \text{Empty}). \lambda(x : (\forall \beta \# \alpha. \text{Int})). 1) [\{l : \text{Int}\}] (\Lambda(\beta \# \{l : \text{Int}\}). 2)$$

The term type-checks in λ^{\parallel} and has type Int . During elaboration, we treat records differently according to where they occur. For the type argument $\{l : \text{Int}\}$, since it is not in a constraint list, we elaborate it normally to $\{l : \text{Int}\}$. For the term argument $(\Lambda(\beta \# \{l : \text{Int}\}). 2)$, since the record $\{l : \text{Int}\}$ appears in a constraint list, we elaborate the term argument to $(\Lambda(\beta * \{l : \perp\}). 2)$. The whole term is then elaborated to

$$E_1 = (\Lambda(\alpha * \top). ((\lambda x. 1) : (\forall (\beta * \alpha). \text{Int}) \rightarrow \text{Int})) \{l : \text{Int}\} (\Lambda(\beta * \{l : \perp\}). 2)$$

However, E_1 fails to type-check in F_i^+ : after type application, we substitute α with the type argument $\{l : \text{Int}\}$ in x 's type $(\forall (\beta * \alpha). \text{Int})$, yielding $(\forall (\beta * \{l : \text{Int}\}). \text{Int})$, whereas the term argument has type $(\forall (\beta * \{l : \perp\}). \text{Int})$, which does not match (and is not a subtype of) the expected parameter type!

The tricky part here is that, for type variables that appear in the constraint list, after type application, the elaborated disjointness constraint contains the original type argument instead of the bottom-elaborated type. In this case, the result type of type application, i.e., $((\forall (\beta * \{l : \text{Int}\}). \text{Int}) \rightarrow \text{Int})$, has $\{l : \text{Int}\}$ instead of $\{l : \perp\}$ in the disjointness constraint.

Apparently we cannot bottom-elaborate every type argument, or otherwise we would lose type information for records. For example, $((\Lambda(\alpha \# \text{Empty}). \lambda(x : \alpha). x) [\{l : \text{Int}\}] \{l = 1\}). l + 1$ should not elaborate to $((\Lambda(\alpha * \top). (\lambda x. x) : \alpha \rightarrow \alpha) \{l : \perp\} \{l = 1\}). l + 1$, which is ill-typed.

Therefore, we *bottom-elaborate record variables that appear in a constraint list*. To this end, we map a record type variable α to a pair of type variables α and α_{\perp} , where α_{\perp} is used in the disjointness constraint. Note that, α_{\perp} is *not* a new sort of type variable—we can use α_1 or α_2 as well – the subscript \perp here is only for readability. The bottom-elaborated type variable α_{\perp} is introduced by an extra type abstraction. While α takes the normal type argument, α_{\perp} takes an extra bottom-elaborated type argument. As an example, the expression ε_1 in Example 8 is elaborated to E'_1 , which type-checks successfully in F_i^+ , where the differences from E_1 are highlighted in gray.

$$E'_1 = (\Lambda(\alpha * \top). \Lambda(\alpha_{\perp} * \top). (\lambda x. 1) : (\forall (\beta * \alpha_{\perp}). \text{Int}) \rightarrow \text{Int}) \{l : \text{Int}\} \{l : \perp\} (\Lambda(\beta * \{l : \perp\}). 2)$$

Intentionally, α_{\perp} is a *subtype* of α , as it always takes bottom-elaborated type arguments that are subtype of the original type arguments. For example, $\{l : \perp\}$ is a subtype of $\{l : \text{Int}\}$. However, the type system is unaware of this observation.

► **Example 9.** Consider the term

$$\varepsilon_2 = \Lambda(\alpha \# \text{Empty}). \Lambda(\beta \# \alpha). \lambda(x : \alpha). \lambda(y : \beta). x \parallel y.$$

Under the current approach, it elaborates to

$$E_2 = \Lambda(\alpha * \top). \Lambda(\alpha_{\perp} * \top). \Lambda(\beta * \alpha_{\perp}). \Lambda(\beta_{\perp} * \alpha_{\perp}). \lambda(x : \alpha). \lambda(y : \beta). x, y$$

$\llbracket t \rrbracket$	$\llbracket \text{Int} \rrbracket$	$=$	Int	$\llbracket r \rrbracket_{\perp}$	$\llbracket \alpha \rrbracket_{\perp}$	$=$	α_{\perp}
	$\llbracket t_1 \rightarrow t_2 \rrbracket$	$=$	$\llbracket t_1 \rrbracket \rightarrow \llbracket t_2 \rrbracket$		$\llbracket \text{Empty} \rrbracket_{\perp}$	$=$	\top
	$\llbracket \forall \alpha \# R. t \rrbracket$	$=$	$\forall(\alpha * \llbracket R \rrbracket). \forall(\alpha_{\perp} * \llbracket R \rrbracket). \llbracket t \rrbracket$		$\llbracket \{l : t\} \rrbracket_{\perp}$	$=$	$\{l : \perp\}$
	$\llbracket \alpha \rrbracket$	$=$	α		$\llbracket r_1 \parallel r_2 \rrbracket_{\perp}$	$=$	$\llbracket r_1 \rrbracket_{\perp} \& \llbracket r_2 \rrbracket_{\perp}$
	$\llbracket \text{Empty} \rrbracket$	$=$	\top	$\llbracket R \rrbracket$	$\llbracket \diamond \rrbracket$	$=$	\top
	$\llbracket \{l : t\} \rrbracket$	$=$	$\{l : \llbracket t \rrbracket\}$		$\llbracket r, R \rrbracket$	$=$	$\llbracket r \rrbracket \& \llbracket r \rrbracket_{\perp} \& \llbracket R \rrbracket$
	$\llbracket r_1 \parallel r_2 \rrbracket$	$=$	$\llbracket r_1 \rrbracket \& \llbracket r_2 \rrbracket$	$\llbracket T \rrbracket$	$\llbracket \diamond \rrbracket$	$=$	\bullet
$\llbracket G \rrbracket$	$\llbracket \diamond \rrbracket$	$=$	\bullet		$\llbracket T, \alpha \# R \rrbracket$	$=$	$\llbracket T \rrbracket, \alpha * \llbracket R \rrbracket, \alpha_{\perp} * \llbracket R \rrbracket$
	$\llbracket G, x : t \rrbracket$	$=$	$\llbracket G \rrbracket, x : \llbracket t \rrbracket$				

■ **Figure 6** Elaboration functions from λ^{ll} to F_i^+ .

However, the merge $x, , y$ fails to type-check, as we do not have the information that $\alpha * \beta$. We only have $\beta * \alpha_{\perp}$ in the context. If the system could know that $\alpha_{\perp} <: \alpha$, then by Lemma 1 we could derive $\beta * \alpha$.

Twisting F_i^+ by adding the axiom $\alpha_{\perp} <: \alpha$ is unsatisfactory, as it complicates the subtyping relation and also significantly affects the metatheory. Our solution is to include both the regularly elaborated types as well as the bottom-elaborated types into the disjointness constraint. In other words, β is disjoint with both α and α_{\perp} . Now ε_2 elaborates to E'_2 , which type-checks successfully in F_i^+ . Note we have also elaborated and bottom-elaborated **Empty**.

$$E'_2 = \Lambda(\alpha * \top \& \top). \Lambda(\alpha_{\perp} * \top \& \top). \Lambda(\beta * \alpha \& \alpha_{\perp}). \Lambda(\beta_{\perp} * \alpha \& \alpha_{\perp}). \lambda x : \alpha. \lambda y : \beta. x, , y$$

4.5 Formal Elaboration

With all the above ideas and observations in mind, we are ready to give a formal account of the elaboration. The elaboration of types is given in Figure 6. We highlight the differences from Figure 5 in grey. There are two ways of elaborating records: $\llbracket r \rrbracket$ (contained in $\llbracket t \rrbracket$) for regular elaboration and $\llbracket r \rrbracket_{\perp}$ for bottom elaboration. In regular elaboration $\llbracket t \rrbracket$, α elaborates to α . Of particular interest is the case of elaborating quantifiers: each quantifier $\forall \alpha \# R. t$ is split into two quantifiers $\forall(\alpha * \llbracket R \rrbracket). \forall(\alpha_{\perp} * \llbracket R \rrbracket). \llbracket t \rrbracket$ in F_i^+ . The relative order of α and α_{\perp} is not important, as long as we respect the order when elaborating type applications. Bottom elaboration $\llbracket r \rrbracket_{\perp}$ elaborates α to α_{\perp} , and $\{l : t\}$ to $\{l : \perp\}$.

When elaborating constraint lists ($\llbracket R \rrbracket$), a record r is elaborated to the intersection of both its regular elaboration and bottom elaboration. Thus if β is compatible with α , then its elaboration β is disjoint with both α and α_{\perp} .

Now let us go back to the gray parts in Figure 4. The major difference from Figure 5 is rule WTT-ALLI and rule WTT-ALLE. In rule WTT-ALLI, we elaborate constrained type abstractions to disjoint type abstractions with two quantifiers, matching the elaboration of constrained quantification. Note that the relative order of α and α_{\perp} should match the order of α and α_{\perp} in elaborating quantifiers. Similarly, in the type application $\varepsilon[r]$ (rule WTT-ALLE), we first elaborate e to E . The elaboration E is then applied to two types $\llbracket r \rrbracket$ and $\llbracket r \rrbracket_{\perp}$, as E has two quantifiers resulting from the elaboration. It is of great importance that the relative order of $\llbracket r \rrbracket$ and $\llbracket r \rrbracket_{\perp}$ should match the order of α and α_{\perp} in elaborating quantifiers. There is a *protocol* that we must follow during elaboration: if α is substituted by $\llbracket r \rrbracket$, then α_{\perp} is substituted by $\llbracket r \rrbracket_{\perp}$.

4.6 Metatheory

Our elaboration enjoys desirable properties. The following lemma states that our elaboration function commutes with substitution, in a slightly involved way:

► **Lemma 10** (Elaboration commutes with substitution). *We have (1) $\llbracket [r/\alpha]t \rrbracket = \llbracket [r]_{\perp}/\alpha_{\perp} \rrbracket \llbracket [r]/\alpha \rrbracket \llbracket t \rrbracket$; (2) $\llbracket [r/\alpha]r_1 \rrbracket_{\perp} = \llbracket [r]_{\perp}/\alpha_{\perp} \rrbracket \llbracket [r]/\alpha \rrbracket \llbracket r_1 \rrbracket_{\perp}$; and (3) $\llbracket [r/\alpha]R \rrbracket = \llbracket [r]_{\perp}/\alpha_{\perp} \rrbracket \llbracket [r]/\alpha \rrbracket \llbracket R \rrbracket$.*

We show key lemmas that *bridge the gap* between row and disjoint polymorphism.

► **Lemma 11** (Type equivalence implies subtyping). *If $t_1 \sim t_2$, then we have $\llbracket t_1 \rrbracket <: \llbracket t_2 \rrbracket$ and $\llbracket t_2 \rrbracket <: \llbracket t_1 \rrbracket$.*

► **Lemma 12** (Compatibility implies disjointness). *If $T \vdash r_1 \# r_2$, then we have: (1) $\llbracket T \rrbracket \vdash \llbracket r_1 \rrbracket * \llbracket r_2 \rrbracket$; (2) $\llbracket T \rrbracket \vdash \llbracket r_1 \rrbracket * \llbracket r_2 \rrbracket_{\perp}$; (3) $\llbracket T \rrbracket \vdash \llbracket r_1 \rrbracket_{\perp} * \llbracket r_2 \rrbracket$; and (4) $\llbracket T \rrbracket \vdash \llbracket r_1 \rrbracket_{\perp} * \llbracket r_2 \rrbracket_{\perp}$.*

► **Lemma 13** (Essence of compatibility). *If $T \vdash r \# \{l : t\}$, then for all A , we have (1) $\llbracket T \rrbracket \vdash \llbracket r \rrbracket * \{l : A\}$; and (2) $\llbracket T \rrbracket \vdash \llbracket r \rrbracket_{\perp} * \{l : A\}$.*

With everything in place, we prove that our elaboration in Figure 4 is type-safe. The reader can refer to our Coq formalization for details.

► **Theorem 14** (Type-safety of elaboration). *If $T; G \vdash \varepsilon : t \rightsquigarrow E$, then $\llbracket T \rrbracket; \llbracket G \rrbracket \vdash E \Rightarrow \llbracket t \rrbracket$.*

Coherence. Because of rule WTT-EQ, a λ^{\parallel} expression can possibly elaborate to many different F_i^+ expressions. For example, the term $\Lambda(\alpha \# \{l : \text{Int}\}). \lambda(x : \alpha). x$ has the following two elaborations E_1 and E_2 (among others). This is the problem of coherence [56]: the meaning of a target program depends on the choice of a particular elaboration typing.

1. $E_1 = \Lambda(\alpha * (\{l : \text{Int}\} \& \{l : \perp\})). \Lambda(\alpha_{\perp} * (\{l : \text{Int}\} \& \{l : \perp\})). \lambda(x : \alpha). x$;
2. $E_2 = (E_1 : \llbracket \forall \alpha \# \{l : \text{Bool}\}. \alpha \rightarrow \alpha \rrbracket) : \llbracket \forall \alpha \# \{l : \text{Int}\}. \alpha \rightarrow \alpha \rrbracket$

To prove that different elaborations are *equivalent*, we utilize the definition of *contextual equivalence*. In particular, we prove that if a λ^{\parallel} expression ε with type t elaborates to two F_i^+ expressions, and these two F_i^+ expressions further elaborate to two F_{co} expressions, then the F_{co} expressions are contextually equivalent.

► **Theorem 15** (Coherence of elaboration). *If $\diamond; \diamond \vdash \varepsilon : t \rightsquigarrow E_1$, and $\diamond; \diamond \vdash \varepsilon : t \rightsquigarrow E_2$, and $\bullet; \bullet \vdash E_1 \Rightarrow \llbracket t \rrbracket \rightsquigarrow e_1$, and $\bullet; \bullet \vdash E_2 \Rightarrow \llbracket t \rrbracket \rightsquigarrow e_2$, then $\bullet; \bullet \vdash e_1 \simeq_{ctx} e_2$.*

5 Encoding Bounded Quantification

This section presents a type-safe and coherent encoding of kernel $F_{<}$: [12] into F_i^+ . This encoding validates the informal observation about the relationship between polymorphic intersection systems and bounded quantification.

5.1 Syntax and Semantics of kernel $F_{<}$:

We start by reviewing the syntax and semantics of kernel $F_{<}$, a polymorphic calculus with bounded quantification. The syntax of $F_{<}$ is given at the top of Figure 7. It is a version of $F_{<}$: extended with records² [10]. In addition to standard System F constructs, types σ include

² We could also encode record types in $F_{<}$, which however is a bit involved.

bounded quantifications $\forall(\alpha <: \tau). \sigma$, which give a *bound* for the type variable; and record types $\{l_1 : \sigma_1, \dots, l_n : \sigma_n\}$, for which we assume all labels are distinct. In addition to standard System F terms, terms ϵ include type abstractions $\Lambda(\alpha <: \sigma). \epsilon$, records $\{l_1 = \epsilon_1, \dots, l_n = \epsilon_n\}$, and projections $\epsilon.l$. Contexts Σ record both the types of term variables, and the bounds of type variables. We use $\Sigma \vdash \sigma$ to mean that a type is well-formed under a context.

Subtyping. The subtyping relation is presented in the middle of Figure 7. Most rules are quite standard. Rule F-SUB-TVAR-BINDS says that a type variable α is a subtype of its bound σ . Rule F-SUB-FORALL, first introduced in Fun [12], requires that the bounds of two quantified types must be identical in order for one to be a subtype of the other. Full $F_{<}$ relaxes this restriction and includes a more powerful formulation where subtyping of quantified types is contravariant in their bounds and covariant in their bodies. We will discuss full $F_{<}$ in Section 6.2. Rules F-SUB-RCDDEPTH, F-SUB-RCDWIDTH, and F-SUB-RCDPERM together form the usual record subtyping.

Typing. The typing rules of $F_{<}$ are shown below the subtyping relation. The reader is advised to ignore the gray parts for now. Most rules are straightforward. Unlike F_i^+ , $F_{<}$ has a subsumption rule (rule F-SUB) for implicit upcasting that can be triggered anywhere during type-checking. Type abstractions are checked by moving their bounds into the context (rule F-TABS), and type applications check that the type being passed satisfies the bound of the corresponding quantifier (rule F-TAPP).

5.2 Elaboration Function

Adapting the encoding from Pierce [48] to our setting, we have

$$\forall(\alpha <: \sigma). \tau \triangleq \forall(\alpha * \top). [\alpha \& \sigma / \alpha] \tau$$

We turn the encoding into an elaboration function. Instead of immediately substituting α with $\alpha \& \sigma$, we collect the bounds $\alpha <: \sigma$ as we traverse the quantifiers, and only substitute when we encounter a type variable α . This strategy is consistent with elaborating types with free type variables. For example, consider the expression $\alpha <: \text{Int} \vdash (\lambda(x : \alpha). x + 1) : \alpha \rightarrow \text{Int}$. This expression type-checks because we have the information $\alpha <: \text{Int}$ in the context so that we can upcast (by rule F-SUB) the type of x to Int when checking $x + 1$. Here it is important to propagate the context information to the type being elaborated. In a fairly standard way, we regard the context as a big binder. Intuitively, if we elaborate α under the context $\alpha <: \text{Int}$, it should give us the same result as if elaborating α inside $\forall(\alpha <: \text{Int}). \alpha$. Therefore, in this case, we substitute α by $\alpha \& \text{Int}$, which yields $x : \alpha \& \text{Int}$, and thus validates $x + 1$.

Formally, type elaboration is denoted as $\llbracket \sigma \rrbracket_{\Sigma} = A$, which reads: under context Σ , type σ elaborates to type A . Elaboration of a closed type is just a special case where the context is empty, i.e., $\llbracket \sigma \rrbracket_{\diamond}$. The full definition is given on the lower left of Figure 7. Most rules are self-explanatory. In particular, bounded quantification elaborates into disjoint quantification by moving the bound information into the context. When elaborating a type variable α , we traverse the context until we find its subtyping constraint $\alpha <: \sigma$, and then we substitute it with an intersection type $\alpha \& \llbracket \sigma \rrbracket_{\Sigma}$.

► **Lemma 16** ($\llbracket \sigma \rrbracket_{\Sigma}$ is total). *If $\Sigma \vdash \sigma$, then there exists a unique type A such that $\llbracket \sigma \rrbracket_{\Sigma} = A$.*

Types	$\sigma, \tau ::= \text{Int} \mid \top \mid \alpha \mid \sigma \rightarrow \tau \mid \forall(\alpha <: \tau). \sigma \mid \{l_1 : \sigma_1, \dots, l_n : \sigma_n\}$
Terms	$\epsilon ::= i \mid \top \mid x \mid \lambda(x : \sigma). \epsilon \mid \epsilon_1 \epsilon_2 \mid \Lambda(\alpha <: \tau). \epsilon \mid \epsilon \sigma \mid \{l_1 = \epsilon_1, \dots, l_n = \epsilon_n\} \mid \epsilon.l$
Value	$v ::= i \mid \top \mid \lambda(x : \sigma). \epsilon \mid \Lambda(\alpha <: \sigma). \epsilon \mid \{l_1 = v_1, \dots, l_n = v_n\}$
Context	$\Sigma ::= \diamond \mid \Sigma, x : \sigma \mid \Sigma, \alpha <: \sigma$

$\Sigma \vdash \sigma <: \tau$	<i>(Subtyping)</i>		
$\frac{\text{F-SUB-REFL} \quad \Sigma \text{ ok} \quad \Sigma \vdash \sigma}{\Sigma \vdash \sigma <: \sigma}$	$\frac{\text{F-SUB-TRANS} \quad \Sigma \vdash \sigma_1 <: \sigma_2 \quad \Sigma \vdash \sigma_2 <: \sigma_3}{\Sigma \vdash \sigma_1 <: \sigma_3}$	$\frac{\text{F-SUB-TOP} \quad \Sigma \text{ ok} \quad \Sigma \vdash \sigma}{\Sigma \vdash \sigma <: \top}$	$\frac{\text{F-SUB-TVAR-BINDS} \quad (\alpha <: \sigma) \in \Sigma}{\Sigma \vdash \alpha <: \sigma}$
$\frac{\text{F-SUB-ARROW} \quad \Sigma \vdash \tau_1 <: \sigma_1 \quad \Sigma \vdash \sigma_2 <: \tau_2}{\Sigma \vdash \sigma_1 \rightarrow \sigma_2 <: \tau_1 \rightarrow \tau_2}$	$\frac{\text{F-SUB-FORALL} \quad \Sigma, \alpha <: \tau \vdash \sigma_1 <: \sigma_2}{\Sigma \vdash \forall(\alpha <: \tau). \sigma_1 <: \forall(\alpha <: \tau). \sigma_2}$	$\frac{\text{F-SUB-RCDDEPTH} \quad \text{for each } i \quad \Sigma \vdash \sigma_i <: \tau_i}{\Sigma \vdash \{l_i : \sigma_i\} <: \{l_i : \tau_i\}}$	
$\frac{\text{F-SUB-RCDWIDTH} \quad \Sigma \vdash \{l_i : \sigma_i^{i \in 1..n+k}\} <: \{l_i : \sigma_i^{i \in 1..n}\}}{\Sigma \vdash \{l_i : \sigma_i^{i \in 1..n+k}\} <: \{l_i : \sigma_i^{i \in 1..n}\}}$	$\frac{\text{F-SUB-RCDPERM} \quad \{l'_j : \tau_j^{j \in 1..n}\} \text{ is a permutation of } \{l_i : \sigma_i^{i \in 1..n}\}}{\Sigma \vdash \{l'_j : \tau_j^{j \in 1..n}\} <: \{l_i : \sigma_i^{i \in 1..n}\}}$		

$\Sigma \vdash \epsilon : \sigma \rightsquigarrow E$	<i>(Typing)</i>	
$\frac{\text{F-TOP} \quad \Sigma \text{ ok}}{\Sigma \vdash \top : \top \rightsquigarrow \top}$	$\frac{\text{F-NAT} \quad \Sigma \text{ ok}}{\Sigma \vdash i : \text{Int} \rightsquigarrow i}$	$\frac{\text{F-VAR} \quad \Sigma \text{ ok} \quad (x : \sigma) \in \Sigma}{\Sigma \vdash x : \sigma \rightsquigarrow x}$
$\frac{\text{F-ARROW} \quad \Sigma, x : \sigma \vdash \epsilon : \tau \rightsquigarrow E}{\Sigma \vdash \lambda(x : \sigma). \epsilon : \sigma \rightarrow \tau \rightsquigarrow (\lambda x. E) : (\llbracket \sigma \rrbracket_\Sigma \rightarrow \llbracket \tau \rrbracket_\Sigma)}$	$\frac{\text{F-SUB} \quad \Sigma \vdash \epsilon : \sigma \rightsquigarrow E \quad \Sigma \vdash \sigma <: \tau}{\Sigma \vdash \epsilon : \tau \rightsquigarrow E : \llbracket \tau \rrbracket_\Sigma}$	
$\frac{\text{F-APP} \quad \Sigma \vdash \epsilon_1 : \sigma \rightarrow \tau \rightsquigarrow E_1 \quad \Sigma \vdash \epsilon_2 : \sigma \rightsquigarrow E_2}{\Sigma \vdash \epsilon_1 \epsilon_2 : \tau \rightsquigarrow E_1 E_2}$	$\frac{\text{F-TABS} \quad \Sigma, \alpha <: \sigma \vdash \epsilon : \tau \rightsquigarrow E}{\Sigma \vdash \Lambda(\alpha <: \sigma). \epsilon : \forall(\alpha <: \sigma). \tau \rightsquigarrow \Lambda(\alpha * \top). E}$	
$\frac{\text{F-RCD} \quad \Sigma \vdash \epsilon_1 : \sigma_1 \rightsquigarrow E_1 \quad \dots \quad \Sigma \vdash \epsilon_n : \sigma_n \rightsquigarrow E_n}{\Sigma \vdash \{l_1 = \epsilon_1, \dots, l_n = \epsilon_n\} : \{l_1 : \sigma_1, \dots, l_n : \sigma_n\} \rightsquigarrow \{l_1 = E_1, \dots, l_n = E_n\}}$		
$\frac{\text{F-PROJ} \quad \Sigma \vdash \epsilon : \{l_1 : \sigma_1, \dots, l : \sigma, \dots, l_n : \sigma_n\} \rightsquigarrow E}{\Sigma \vdash \epsilon.l : \sigma \rightsquigarrow (E : \llbracket \{l : \sigma\} \rrbracket_\Sigma).l}$	$\frac{\text{F-TAPP} \quad \Sigma \vdash \epsilon : \forall(\alpha <: \tau_1). \tau_2 \rightsquigarrow E \quad \Sigma \vdash \sigma <: \tau_1}{\Sigma \vdash \epsilon \sigma : [\sigma/\alpha]\tau_2 \rightsquigarrow (E \llbracket \sigma \rrbracket_\Sigma) : (\llbracket [\sigma/\alpha]\tau_2 \rrbracket_\Sigma)}$	

$\llbracket \sigma \rrbracket_\Sigma$	$\llbracket \text{Int} \rrbracket_\Sigma = \text{Int}$	$\llbracket \Sigma \rrbracket$	$\llbracket \diamond \rrbracket = \bullet$
	$\llbracket \top \rrbracket_\Sigma = \top$		$\llbracket \Sigma, \alpha <: \sigma \rrbracket = \llbracket \Sigma \rrbracket, \alpha * \top$
	$\llbracket (\sigma \rightarrow \tau) \rrbracket_\Sigma = \llbracket \sigma \rrbracket_\Sigma \rightarrow \llbracket \tau \rrbracket_\Sigma$		$\llbracket \Sigma, x : \sigma \rrbracket = \llbracket \Sigma \rrbracket$
	$\llbracket (\{l_1 : \sigma_1, \dots, l_n : \sigma_n\}) \rrbracket_\Sigma = \{l_1 : \llbracket \sigma_1 \rrbracket_\Sigma\} \& \dots \& \{l_n : \llbracket \sigma_n \rrbracket_\Sigma\}$		
	$\llbracket \alpha \rrbracket_{(\Sigma, x : \sigma)} = \llbracket \alpha \rrbracket_\Sigma$		
	$\llbracket \alpha \rrbracket_{(\Sigma, \beta <: \sigma)} = \llbracket \alpha \rrbracket_\Sigma$	$\llbracket \llbracket \Sigma \rrbracket \rrbracket$	$\llbracket \diamond \rrbracket = \bullet$
	$\llbracket \alpha \rrbracket_{(\Sigma, \alpha <: \sigma)} = \alpha \& \llbracket \sigma \rrbracket_\Sigma$		$\llbracket \llbracket \Sigma, \alpha <: \sigma \rrbracket \rrbracket = \llbracket \llbracket \Sigma \rrbracket \rrbracket$
	$\llbracket \forall(\alpha <: \sigma). \tau \rrbracket_\Sigma = \forall(\alpha * \top). \llbracket \tau \rrbracket_{\Sigma, \alpha <: \sigma}$		$\llbracket \llbracket \Sigma, x : \sigma \rrbracket \rrbracket = \llbracket \llbracket \Sigma \rrbracket, x : \llbracket \sigma \rrbracket_\Sigma \rrbracket$

■ **Figure 7** Syntax, subtyping, typing and elaboration of kernel $F_{<}$.

We now lift the elaboration function to contexts, given on the lower right of Figure 7. $\llbracket \Sigma \rrbracket$ elaborates a $F_{<}:$ context to a F_i^+ type context, in which subtyping constraints $\alpha <: \sigma$ of type variables are elaborated to disjointness constraints $\alpha * \top$ and all term variables are ignored. $\llbracket \Sigma \rrbracket$ elaborates a $F_{<}:$ context to a F_i^+ term context, in which all type variables are ignored and the types of term variables are elaborated under the prefix context.

5.3 Type-directed Elaboration

An intuitive elaboration scheme of expressions is to simply apply the elaboration function to types. For example, under context Σ , if ϵ elaborates to E , then type applications $\epsilon \sigma$ elaborates to $E \llbracket \sigma \rrbracket_{\Sigma}$. Now let us consider an example.

► **Example 17.** Consider a $F_{<}:$ judgment

$$\beta <: \text{Int} \vdash (\Lambda(\alpha <: \top). \lambda(x : \alpha). x) \beta : \beta \rightarrow \beta$$

Here the type application type-checks because by rule F-SUB-TOP we have $\beta <: \top$. If we elaborate $\epsilon \sigma$ to $E \llbracket \sigma \rrbracket_{\Sigma}$ directly, the resulting expression is

$$(\Lambda(\alpha * \top). (\lambda x. x) : (\alpha \& \top) \rightarrow (\alpha \& \top)) (\beta \& \text{Int})$$

Note that as F_i^+ does not have annotated abstractions, we put the elaborated arrow type as the type annotation. Following the typing rule of F_i^+ , we can infer the type of this expression:

$$\beta * \top; \bullet \vdash (\Lambda(\alpha * \top). ((\lambda x. x) : (\alpha \& \top) \rightarrow (\alpha \& \top)) (\beta \& \text{Int})) \Rightarrow (\beta \& \text{Int} \& \top) \rightarrow (\beta \& \text{Int} \& \top)$$

However, the expected result type $\beta \rightarrow \beta$ elaborates to

$$(\beta \& \text{Int}) \rightarrow (\beta \& \text{Int})$$

Now we get a mismatch between the actual type $((\beta \& \text{Int} \& \top) \rightarrow (\beta \& \text{Int} \& \top))$ and the expected type $((\beta \& \text{Int}) \rightarrow (\beta \& \text{Int}))$ of the expression!

Fortunately, in this particular example, we can prove that the actual type and the expected type are subtypes of each other, i.e., they are isomorphic. Why is that true? Recall that we have $\beta <: \top$, which after elaboration gives us $(\beta \& \text{Int}) <: \top$. Therefore we can show that the following two subtyping instances are valid: (1) $(\beta \& \text{Int} \& \top) \rightarrow (\beta \& \text{Int} \& \top) <: (\beta \& \text{Int}) \rightarrow (\beta \& \text{Int})$; and (2) $(\beta \& \text{Int}) \rightarrow (\beta \& \text{Int}) <: (\beta \& \text{Int} \& \top) \rightarrow (\beta \& \text{Int} \& \top)$.

More generally, we prove that elaboration commutes with substitution, yielding isomorphic types. Consider that under the context Σ , we have a type application $\epsilon \sigma$, where ϵ has type $\forall(\alpha <: \tau_1). \tau_2$, and in order for it to type-check, we have $\sigma <: \tau_1$. The expected type we want of the expression is the elaboration of the $F_{<}:$ typing result, i.e., $\llbracket ([\sigma/\alpha]\tau_2) \rrbracket_{\Sigma}$. The actual type is the result of feeding the elaborated argument $\llbracket \sigma \rrbracket_{\Sigma}$ to the elaborated quantification $\llbracket \forall(\alpha <: \tau_1). \tau_2 \rrbracket_{\Sigma}$, i.e., $\llbracket [\sigma]_{\Sigma}/\alpha \rrbracket (\llbracket \tau_2 \rrbracket_{(\Sigma, \alpha <: \tau_1)})$.

► **Lemma 18 (Elaboration commutes with substitution).** *Given $\Sigma \vdash \sigma <: \tau_1$, we have (1) $\llbracket [\sigma/\alpha]\tau_2 \rrbracket_{\Sigma} <: \llbracket [\sigma]_{\Sigma}/\alpha \rrbracket (\llbracket \tau_2 \rrbracket_{(\Sigma, \alpha <: \tau_1)})$; and (2) $\llbracket [\sigma]_{\Sigma}/\alpha \rrbracket (\llbracket \tau_2 \rrbracket_{(\Sigma, \alpha <: \tau_1)}) <: \llbracket ([\sigma/\alpha]\tau_2) \rrbracket_{\Sigma}$.*

Note that *the elaboration scheme slightly varies depending on the type semantics of the target intersection type calculi*. It is a desirable property that typing should be preserved after elaboration, i.e., the elaborated expression should have the corresponding elaborated type. For languages with an implicit subsumption rule (e.g., rule F-SUB in kernel $F_{<}:$), Lemma 18 can implicitly upcast the actual type to the expected type, and thus validates the

$$\begin{array}{ccc}
\text{kernel } F_{<} & (\Lambda(\alpha <: \text{Int}). \lambda(x : \alpha). 1) \text{Int} & \longrightarrow & \lambda(x : \text{Int}). 1 \\
& \Downarrow & & \Downarrow \\
F_i^+ & ((\Lambda(\alpha * \top). ((\lambda x. 1) : \alpha \& \text{Int} \rightarrow \text{Int})) \text{Int}) : \text{Int} \rightarrow \text{Int} & & (\lambda x. 1) : \text{Int} \rightarrow \text{Int} \\
& \Downarrow & & \Downarrow \\
F_{co} & ((\text{id}, \text{id}) \rightarrow \text{id}) ((\Lambda \alpha. \lambda x. 1) \text{Int}) & \longrightarrow & ((\text{id}, \text{id}) \rightarrow \text{id}) (\lambda x. 1) \simeq_{ctx} \lambda x. 1
\end{array}$$

■ **Figure 8** Key idea of simulation illustrated with an example.

intuitive elaboration of the type applications. For languages with *explicit* subsumption rules (e.g., rule T-SUB in F_i^+), to remedy this situation, we need to *annotate the expression with the expected type* to explicitly upcast the type. Concretely, in this example, the elaborated expression, with the added annotation highlighted in grey, will be:

$$((\Lambda(\alpha * \top). (\lambda x. x) : (\alpha \& \top) \rightarrow (\alpha \& \top)) (\beta \& \text{Int})) : (\beta \& \text{Int}) \rightarrow (\beta \& \text{Int})$$

Finally, we can go back and consider the elaboration of expressions in the grey part of Figure 7. Most of the elaboration rules are self-explanatory. In particular, in rule F-TAPP, type applications $\epsilon \sigma$ elaborates to $(E \llbracket \sigma \rrbracket_{\Sigma}) : \llbracket ([\sigma/\alpha] \tau_2) \rrbracket_{\Sigma}$.

5.4 Metatheory

Now that we have everything in place, we are ready to prove that our elaboration is sound.

► **Theorem 19** (Type-safety of elaboration). *If $\Sigma \vdash \epsilon : \sigma \rightsquigarrow E$, then $\llbracket \Sigma \rrbracket; \llbracket \Sigma \rrbracket \vdash E \Rightarrow \llbracket \sigma \rrbracket_{\Sigma}$.*

However, due to the implicit upcasting (rule F-SUB), a $F_{<}$ expression can possibly elaborate to many different ones in F_i^+ . For example, consider $(\lambda(x : \top). 2) 1$. Two elaborations (among others) are (1) $((\lambda x. 2) : \top \rightarrow \text{Int}) (1 : \top)$; and (2) $((\lambda x. 2) : \top \rightarrow \text{Int}) : \text{Int} \rightarrow \text{Int} 1$. Therefore, we prove that different elaborations lead to *contextually equivalent* results.³

► **Theorem 20** (Coherence of elaboration). *If $\diamond \vdash \epsilon : \sigma \rightsquigarrow E_1$, and $\diamond \vdash \epsilon : \sigma \rightsquigarrow E_2$, and $\bullet; \bullet \vdash E_1 \Rightarrow \llbracket \sigma \rrbracket_{\diamond} \rightsquigarrow e_1$, and $\bullet; \bullet \vdash E_2 \Rightarrow \llbracket \sigma \rrbracket_{\diamond} \rightsquigarrow e_2$, then $\bullet; \bullet \vdash e_1 \simeq_{ctx} e_2$.*

We also prove a weaker *simulation* result⁴: if the standard direct operational semantics of kernel $F_{<}$ produces $\epsilon_1 \longrightarrow \epsilon_2$, and ϵ_2 elaborates to E_2 in F_i^+ , which in turn elaborates to e_2 in F_{co} , then ϵ_1 elaborates to E_1 in F_i^+ , which in turn elaborates to e_1 in F_{co} , and $e_1 \longrightarrow e'_1$, where e'_1 and e_2 are contextually equivalent. The lemma is weaker in the sense that e'_1 and e_2 are not syntactically equivalent. Given the coherence lemmas of F_i^+ and of the elaboration, it is no surprise that here contextual equivalence takes the place of the syntactic equivalence, as explicit upcasting generates coercions, which may break syntactic equivalence. As an example, consider Figure 8, where e_1 steps to an expression $e'_1 = ((\text{id}, \text{id}) \rightarrow \text{id}) (\lambda x. 1)$ that is contextually equivalent to $e_2 = \lambda x. 1$.

³ One restriction in Bi et al. [7] is that due to the well-foundedness issue, the logical relation of F_i^+ is defined only for its *predicative* subset, where type arguments in type applications can only be monotypes. Since our proof is built upon the logical relation of F_i^+ , Theorem 20 is restricted to predicative subset of kernel $F_{<}$ as well. If the well-foundedness of impredicative F_i^+ is recovered, e.g., by employing step-indexing logical relations [1], we expect that our proof remains valid.

⁴ Note that λ^{\parallel} does not provide a semantics [34], so we did not discuss the operational semantics in Section 4. If λ^{\parallel} had a operational semantics, we believe a similar theorem would apply.

► **Theorem 21** (Simulation). *If $\epsilon_1 \longrightarrow \epsilon_2$, and $\diamond \vdash \epsilon_2 : \sigma \rightsquigarrow E_2$, and $\bullet; \bullet \vdash E_2 \Rightarrow \llbracket \sigma \rrbracket_\diamond \rightsquigarrow e_2$, then there exist E_1, e_1, e'_1 such that $\diamond \vdash \epsilon_1 : \sigma \rightsquigarrow E_1$, and $\bullet; \bullet \vdash E_1 \Rightarrow \llbracket \sigma \rrbracket_\diamond \rightsquigarrow e_1$, and $e_1 \longrightarrow e'_1$, where $\bullet; \bullet \vdash e'_1 \simeq_{ctx} e_2$.*

The detailed paper proof of this lemma is given in Appendix D. This lemma requires a generalized logical equivalence for F_i^+ , which is not yet supported in the current Coq framework. Therefore we only present the paper proof. If the Coq framework of F_i^+ is generalized, we expect that the lemma can be proved in Coq.

6 Discussion

In this section we discuss some possible paths for further exploration.

6.1 Variants of Row Polymorphism

According to Rémy [53], record calculi can typically be categorized into two groups based on how they support the extension operation: the *strict* group does not allow duplicate labels, while the *free* group does. We have already shown that F_i^+ supports λ^{\parallel} , a calculus in the strict group, with a more fine-grained control as disjointness allows duplicate labels as long as their types are disjoint. λ^{TIR} [60] is another calculus from the strict group, which introduces type-indexed rather than label-indexed rows, and uses *membership constraints* to avoid conflicts. To distinguish types and row, λ^{TIR} incorporates a kind system that distinguishes rows from types. We believe that F_i^+ could also serve as a target for λ^{TIR} , as type-indexed rows are closely related to disjoint intersections. Thus an elaboration from λ^{TIR} to F_i^+ is interesting future work.

For the free group, there are two different approaches for extension: previous fields are always retained, and record projections always select the first matching label [35]; or the extension overwrites the field if it is already present [5, 53, 11]. The former system suffers from the similar issue of *ambiguity*, as records can be extended with the same label even when types are overlapping, which violates the essence of disjointness. For the latter system, essentially F_i^+ is capable to encode the extension operation in a different form. Consider a function that overwrites (\leftarrow) the label l in a record by incrementing the original value [11]:

$$inc = \Lambda \alpha <: \{l : \text{Int}\}. \lambda(x : \alpha). x \leftarrow \{l = x.l + 1\}$$

In F_i^+ , we can define

$$inc' = \Lambda(\alpha * \{l : \text{Int}\}). \lambda(x : \alpha \& \{l : \text{Int}\}). (x : \alpha, \{l = (x : \{l : \text{Int}\}).l + 1\})$$

There are two differences. Firstly, the type arguments to the two functions are different: inc expects a type argument which includes $\{l : \text{Int}\}$, while inc' expects a type argument which excludes $\{l : \text{Int}\}$, and $\{l : \text{Int}\}$ is later recovered in x 's type by an intersection type. This explains a more involved encoding. Secondly, the term arguments to the two functions are also different: inc accepts arguments that have exactly one l label with type Int , while inc' can accept arguments of type $\{l : \text{Int}\} \& \{l : \text{Bool}\}$. This again manifests the fine-grained control of disjointness. That being said, we have not studied nor formalized the encoding.

Type-inference. The focus of our work is languages that have more modest goals in terms of type-inference. Note that neither λ^{\parallel} or F_i^+ address sophisticated type-inference. We focus on languages with subtyping, including TypeScript, Ceylon, Scala or Flow. Languages like

Racket also include a variant of row polymorphism, without full-type inference to model powerful OOP features [61]. Many other row type systems [53, 64, 63, 35] support type inference. For the future, we wish to investigate whether a disjoint polymorphic calculus offering similar type inference can model calculi with row polymorphism and type inference. We believe that several ideas employed in work on type inference for row polymorphism can be adapted to a setting with disjoint polymorphism.

6.2 Variants of Bounded Quantification

Full $F_{<}$: [23] includes a more powerful formulation of subtyping for universal quantification (rule F-SUB-FORALLALT), which is contravariant in the bound types and covariant in the body types. However, this subtyping rule renders subtyping in full $F_{<}$: undecidable [49].

$$\frac{\Sigma \vdash \tau_2 <: \tau_1 \quad \Sigma, \alpha <: \tau_2 \vdash \sigma_1 <: \sigma_2}{\Sigma \vdash \forall(\alpha <: \tau_1). \sigma_1 <: \forall(\alpha <: \tau_2). \sigma_2} \text{F-SUB-FORALLALT}$$

Moreover, this rule breaks the encoding. Consider the example [48]:

$$\diamond \vdash \forall(\alpha <: \top). \alpha <: \forall(\alpha <: \text{Int}). \alpha$$

which elaborates to a non-derivable F_i^+ judgment

$$\bullet \vdash \forall(\alpha * \top). \alpha \& \top <: \forall(\alpha * \top). \alpha \& \text{Int}$$

since $\alpha * \top \vdash \alpha \& \top <: \alpha \& \text{Int}$ is not true.

One possible solution is to adopt a more powerful subtyping relation in the target calculus, where a polymorphic type is a subtype of one type if the first has more instances [45]. For example, the following judgment holds true, as α can be instantiated to Int to get $\text{Int} \rightarrow \text{Int}$:

$$\forall \alpha. \alpha \rightarrow \alpha <: \text{Int} \rightarrow \text{Int}$$

Then the judgment $\bullet \vdash \forall(\alpha * \top). \alpha \& \top <: \forall(\alpha * \top). \alpha \& \text{Int}$ is derivable. After we skolemise the type variable α in the right hand side, we can instantiate α in the left hand side by $\alpha \& \text{Int}$ to get $\alpha * \top \vdash \alpha \& \text{Int} \& \top <: \alpha \& \text{Int}$.

Interestingly, such subtyping is usually *predicative*, i.e., universal quantifications can only be instantiated with monotypes; or otherwise it is undecidable. Thus if the bounds can only be monotypes, it may be the case that a target calculus with the more powerful subtyping rule can encode the predicative version of full $F_{<}$.

6.3 Variants of Intersection Type Systems

λ^{\parallel} is encodable into intersection type systems that feature the merge operator, unrestricted intersection types, polymorphism and guarantee coherence through constraints similar to compatibility or disjointness. This currently only applies to F_i^+ . Some intersection type systems [28, 6, 46] only support simple record types. While Alpuim and Oliveira [2] do support polymorphism, they only allow intersection types between disjoint types. Hence, our elaboration of constraint lists to $\llbracket r \rrbracket \& \llbracket r \rrbracket_{\perp}$ is rejected as $\llbracket r \rrbracket$ and $\llbracket r \rrbracket_{\perp}$ may not be disjoint.

Kernel $F_{<}$: is encodable for intersection type systems that feature polymorphism and unrestricted intersection types. For example, a similar encoding might be applicable to other intersection type systems [17, 19]. Interestingly, the behavior of elaborated expressions varies according to the type semantics of the target. Consider a function f of type $\forall(\alpha <: \text{Int}). \alpha \rightarrow \alpha$, which, based on the encoding, elaborates to $\forall \alpha. \alpha \& \text{Int} \rightarrow \alpha \& \text{Int}$. The original type expects a type argument which is a subtype of Int ; while in the intersection type system, the elaborated

type can take any type argument, e.g., `Bool`, and then expect a term argument of type `Int & Bool`. In intersection type systems (e.g., [43]) where `Int & Bool` is uninhabited (equivalent to the bottom type), `f Bool` can take nothing. Yet, in calculi with the merge operator, we can have, e.g., `f Bool (1, , True)`.

7 Related Work

Bounded quantification and intersection types. The language `Fun` [12] introduced bounded quantification. Bounded quantification is later extended with extensible records [10, 11], recursively defined types [9] and session types [25, 33] among other extensions. The full variant of $F_{<}$ [23] (see also Section 6.2) is proved to be undecidable [49]. The kernel `Fun` variant [12], which restricts the subtyping of bounds to be invariant, is decidable.

Pierce [48] proposed the encoding of bounded quantification in terms of intersection types in an informal discussion, which is the main inspiration of our Section 5. Castagna and Xu [19] mentioned in a footnote that a type variable α bounded by a type σ can be encoded by replacing every occurrence of α by $\beta \wedge \sigma$ where β is a fresh unbounded variable. Castagna et al. [17] further mentioned that the possible instantiation of a type variable α with an upper bound σ and a lower bound τ is equivalent to the possible instantiation of $(\tau \vee \beta) \wedge \sigma$. Dolan and Mycroft [26] used a similar encoding as one of the main ingredients of the biunification algorithm: $\alpha <: \sigma^-$ (where types have polarity) implies the bisubstitution $\theta = [(\mu^- \beta. \alpha \sqcap [\beta/\alpha^-](\sigma^-))/\alpha^-]$, which by unrolling implies that $\theta(\alpha^-) = \alpha \sqcap \theta(\sigma^-)$. The idea of encoding bounded quantification using intersection types is not new. However, as far as we know, we are the first to formalize an elaboration and study the metatheory from a calculus with bounded quantification into a calculus with intersection types and polymorphism. This contrasts with the previous informal discussions, which have only shown a few concrete examples of programs that could be manually translated (or not).

Row calculi and intersection types. Along the way we have mentioned many row calculi [35, 5, 53, 11, 64, 63]. Reynolds [57] developed an encoding of simple records in terms of intersection types and his merge construct. Similar ideas had been applied by more recent work on intersection types with a merge operator [28, 6, 2]. Alpuim and Oliveira [2] showed informally that many features of row polymorphism can be simulated with disjoint polymorphism. However, their system is limiting for the encoding in Section 4.4.

Intersection types and the merge operator. The F_i^+ calculus follows from a line of work on intersection types with a merge operator. The programming language `Forsythe` [57, 55] includes a merge operator. However, several restrictions were imposed to make the merge operator coherent [56]. For example, merging two functions is forbidden. Castagna et al. [14] studied a special merge operator that only works on functions. Dunfield [28] proposed a calculus with unrestricted intersection types and unrestricted merges. However his calculus loses coherence. For example, `1, , 2` could elaborate to `1` or `2`. Pierce [48] proposed a primitive function `glue`, similar to unrestricted merges. Oliveira et al. [46] proposed disjoint intersection types and disjoint merges to recover syntactic coherence. Later this approach was extended with *disjoint polymorphism* [2]. Bi et al. [6] support unrestricted intersection types and disjoint merges, based on a novel semantic coherence approach in terms of contextual equivalence, which is later extended to support polymorphic types [7].

Other work on intersection types includes refinement intersections [24, 27]; set theoretical foundation for type connectives including intersections, unions and negations [16, 15, 17, 19]; and the DOT calculus, which aims at providing a foundational calculus for Scala that

incorporates features including intersection types [3, 58]. In those calculi, intersection types only increase the expressiveness of types, but not the expressiveness of terms. For example, the intersection type $\text{Int} \& \text{Bool}$ is uninhabited. The type system of Ceylon [43] exploits this fact and considers any intersection of such *disjoint* types equivalent to the bottom type (\perp).

8 Conclusion and Future Work

We have presented the elaboration from kernel $F_{<}$ and λ^{\parallel} to F_i^+ , and showed that disjoint polymorphism is powerful enough to encode essential aspects of bounded quantification and row polymorphism, which is useful for economy of theory and implementation. The elaboration from kernel $F_{<}$ identifies one encodable fragment of $F_{<}$, and thus validates the previous informal observation by Pierce. The elaboration from λ^{\parallel} to F_i^+ reveals the essence of constrained quantification from the point of view of disjointness. As for future work, we plan to study the encoding of other variants of $F_{<}$, as well as other row calculi. We also plan to study type inference of F_i^+ .

References

- 1 Amal Ahmed. Step-indexed syntactic logical relations for recursive and quantified types. In *European Symposium on Programming (ESOP)*, 2006.
- 2 João Alpuim, Bruno C. d. S. Oliveira, and Zhiyuan Shi. Disjoint polymorphism. In *European Symposium on Programming (ESOP)*, 2017.
- 3 Nada Amin, Adriaan Moors, and Martin Odersky. Dependent object types. In *Workshop on Foundations of Object-Oriented Languages*, 2012.
- 4 Henk Barendregt, Mario Coppo, and Mariangiola Dezani-Ciancaglini. A filter lambda model and the completeness of type assignment. *The journal of symbolic logic*, 48(04):931–940, 1983.
- 5 Bernard Berthomieu and Camille Le Monies De Sagazan. A calculus of tagged types, with applications to process languages. *Types for Program Analysis*, page 1, 1995.
- 6 Xuan Bi, Bruno C. d. S. Oliveira, and Tom Schrijvers. The essence of nested composition. In *European Conference on Object-Oriented Programming (ECOOP)*, 2018.
- 7 Xuan Bi, Ningning Xie, Bruno C. d. S. Oliveira, and Tom Schrijvers. Distributive disjoint polymorphism for compositional programming. In *European Symposium on Programming (ESOP)*, 2019.
- 8 Gilad Bracha. *The programming language jigsaw: mixins, modularity and multiple inheritance*. PhD thesis, Dept. of Computer Science, University of Utah, 1992.
- 9 Peter Canning, William Cook, Walter Hill, Walter Olthoff, and John C Mitchell. F-bounded polymorphism for object-oriented programming. In *FPCA*, volume 89, pages 273–280, 1989.
- 10 Luca Cardelli. *Extensible records in a pure calculus of subtyping*. Digital. Systems Research Center, 1992.
- 11 Luca Cardelli and John C Mitchell. Operations on records. In *International Conference on Mathematical Foundations of Programming Semantics*, 1989.
- 12 Luca Cardelli and Peter Wegner. On understanding types, data abstraction, and polymorphism. *ACM Computing Surveys*, 17(4):471–523, 1985.
- 13 Felice Cardone. Relational semantics for recursive types and bounded quantification. In *International Colloquium on Automata, Languages, and Programming*, pages 164–178. Springer, 1989.
- 14 Giuseppe Castagna, Giorgio Ghelli, and Giuseppe Longo. A calculus for overloaded functions with subtyping. In *Conference on LISP and Functional Programming*, 1992.
- 15 Giuseppe Castagna, Kim Nguyen, Zhiwu Xu, and Pietro Abate. Polymorphic functions with set-theoretic types: part 2: local type inference and type reconstruction. In *Principles of Programming Languages (POPL)*, 2015.

- 16 Giuseppe Castagna, Kim Nguyen, Zhiwu Xu, Hyeonseung Im, Sergueï Lenglet, and Luca Padovani. Polymorphic functions with set-theoretic types: part 1: syntax, semantics, and evaluation. In *Principles of Programming Languages (POPL)*, 2014.
- 17 Giuseppe Castagna, Tommaso Petrucciani, and Kim Nguyen. Set-theoretic types for polymorphic variants. In *International Conference on Functional Programming (ICFP)*, 2016.
- 18 Giuseppe Castagna and Benjamin C Pierce. Decidable bounded quantification. In *Principles of Programming Languages (POPL)*, 1994.
- 19 Giuseppe Castagna and Zhiwu Xu. Set-theoretic foundation of parametric polymorphism and subtyping. In *International Conference on Functional Programming (ICFP)*, 2011.
- 20 C. Chambers, D. Ungar, B.W. Chang, and U. Hölzle. Parents are shared parts of objects: Inheritance and encapsulation in SELF. *Lisp and Symbolic Computation*, 4(3):207–222, 1991.
- 21 Adriana B Compagnoni and Benjamin C Pierce. Higher-order intersection types and multiple inheritance. *Mathematical Structures in Computer Science (MSCS)*, 6(5):469–501, 1996.
- 22 Mario Coppo, Mariangiola Dezani-Ciancaglini, and Patrick Sallé. Functional characterization of some semantic equalities inside λ -calculus. In *International Colloquium on Automata, Languages, and Programming*, pages 133–146. Springer, 1979.
- 23 Pierre-Louis Curien and Giorgio Ghelli. Coherence of subsumption, minimum typing and type-checking in $f\leq$. *Mathematical structures in computer science*, 2(1):55–91, 1992.
- 24 Rowan Davies. *Practical refinement-type checking*. PhD thesis, School of Computer Science, Carnegie Mellon University, 2005.
- 25 Mariangiola Dezani-Ciancaglini, Elena Giachino, Sophia Drossopoulou, and Nobuko Yoshida. Bounded session types for object oriented languages. In *Formal Methods for Components and Objects*, pages 207–245. Springer, 2007.
- 26 Stephen Dolan and Alan Mycroft. Polymorphism, subtyping, and type inference in mlsb. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017*, pages 60–72, New York, NY, USA, 2017. ACM. doi:10.1145/3009837.3009882.
- 27 Joshua Dunfield. Refined typechecking with stardust. In *PLPV*, 2007.
- 28 Joshua Dunfield. Elaborating intersection and union types. *Journal of Functional Programming (JFP)*, 24(2-3):133–165, 2014.
- 29 Erik Ernst. Family polymorphism. In *European Conference on Object-Oriented Programming (ECOOP)*, 2001.
- 30 Erik Ernst. The expression problem, scandinavian style. *On Mechanisms For Specialization*, page 27, 2004.
- 31 Facebook. Flow. <https://flow.org/>, 2014.
- 32 Matthew Flatt, Robert Bruce Findler, and Matthias Felleisen. Scheme with classes, mixins, and traits. In *Programming Languages and Systems (APLAS)*, 2006.
- 33 Simon J Gay. Bounded polymorphism in session types. *Mathematical Structures in Computer Science*, 18(5):895–930, 2008.
- 34 Robert Harper and Benjamin Pierce. A record calculus based on symmetric concatenation. In *Principles of Programming Languages (POPL)*, 1991.
- 35 Daan Leijen. Extensible records with scoped labels. *Trends in Functional Programming*, 5:297–312, 2005.
- 36 Daan Leijen. Type directed compilation of row-typed algebraic effects. In *Principles of Programming Languages (POPL)*, 2017.
- 37 Sam Lindley and James Cheney. Row-based effect types for database integration. In *Proceedings of the 8th ACM SIGPLAN workshop on Types in language design and implementation*, pages 91–102. ACM, 2012.
- 38 Sam Lindley and J Garrett Morris. Lightweight functional session types. *Behavioural Types: from Theory to Tools*. River Publishers, pages 265–286, 2017.
- 39 Simon Martini. Bounded quantifiers have interval models. In *Proceedings of the 1988 ACM conference on LISP and functional programming*, pages 164–173. ACM, 1988.
- 40 Microsoft. Typescript. <https://www.typescriptlang.org/>, 2012.

- 41 Microsoft. <https://www.typescriptlang.org/docs/handbook/advanced-types.html>, 2019. Online; accessed 16 June 2019.
- 42 J. Garrett Morris and James McKinna. Abstracting extensible data types: or, rows by any other name. In *Principles of Programming Languages (POPL)*, 2019.
- 43 Fabian Muehlboeck and Ross Tate. Empowering union and intersection types with integrated subtyping. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2018.
- 44 Martin Odersky, Philippe Altherr, Vincent Cremet, Burak Emir, Sebastian Maneth, Stéphane Micheloud, Nikolay Mihaylov, Michel Schinz, Erik Stenman, and Matthias Zenger. An overview of the scala programming language. Technical report, EPFL, 2004.
- 45 Martin Odersky and Konstantin Läufer. Putting type annotations to work. In *Symposium on Principles of Programming Languages (POPL)*, 1996.
- 46 Bruno C. d. S. Oliveira, Zhiyuan Shi, and João Alpuim. Disjoint intersection types. In *International Conference on Functional Programming (ICFP)*, 2016.
- 47 Bruno C. d. S. Oliveira, Tijs Van Der Storm, Alex Loh, and William R Cook. Feature-oriented programming with object algebras. In *European Conference on Object-Oriented Programming (ECOOP)*, 2013.
- 48 Benjamin C Pierce. *Programming with intersection types and bounded polymorphism*. PhD thesis, University of Pennsylvania, 1991.
- 49 Benjamin C Pierce. Bounded quantification is undecidable. *Information and Computation*, 112(1):131–165, 1994.
- 50 Benjamin C Pierce and David N Turner. Local type argument synthesis with bounded quantification. Technical report, Technical Report 495, Computer Science Department, Indiana University, 1997.
- 51 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.
- 52 Redhat. Ceylon. <https://ceylon-lang.org/>, 2011.
- 53 Didier Rémy. *Type inference for records in a natural extension of ML*. Theoretical Aspects Of Object-Oriented Programming. Types, Semantics and . . . , 1993.
- 54 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, OOPSLA '14*, page 377–395, New York, NY, USA, 2014. Association for Computing Machinery. doi:10.1145/2660193.2660237.
- 55 John C Reynolds. Preliminary design of the programming language forsythe. Technical report, Carnegie Mellon University, 1988.
- 56 John C. Reynolds. The coherence of languages with intersection types. In *Lecture Notes in Computer Science (LNCS)*, pages 675–700. Springer Berlin Heidelberg, 1991.
- 57 John C Reynolds. Design of the programming language forsythe. In *ALGOL-like languages*, pages 173–233. Birkhauser Boston Inc., 1997.
- 58 Tiark Rompf and Nada Amin. Type soundness for dependent object types (DOT). In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2016.
- 59 Patrick Salle. Une extension de la theorie des types en lambda-calcul. In *Proceedings of the Fifth Colloquium on Automata, Languages and Programming*, pages 398–410, London, UK, UK, 1978. Springer-Verlag.
- 60 Mark Shields and Erik Meijer. Type-indexed rows. In *Proceedings of the 28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '01*, pages 261–275, New York, NY, USA, 2001. ACM. doi:10.1145/360204.360230.
- 61 Asumu Takikawa, T. Stephen Strickland, Christos Dimoulas, Sam Tobin-Hochstadt, and Matthias Felleisen. Gradual typing for first-class classes. In *Object-oriented Programming: Systems, Languages and Applications (OOPSLA)*, 2012.
- 62 Philip Wadler. The expression problem. *Java-genericity mailing list*, 1998.

27:30 Row and Bounded Polymorphism via Disjoint Polymorphism

- 63 Mitchell Wand. Complete type inference for simple objects. In *Symposium on Logic in Computer Science (LICS)*, 1987.
- 64 Mitchell Wand. Type inference for record concatenation and multiple inheritance. In *Symposium on Logic in Computer Science (LICS)*, 1989.
- 65 Mathias Zenger and Martin Odersky. Independently extensible solutions to the expression problem. In *Foundations of Object-Oriented Languages*, 2005.

A Trusted Infrastructure for Symbolic Analysis of Event-Driven Web Applications

Gabriela Sampaio

Imperial College London, United Kingdom
g.sampaio17@imperial.ac.uk

José Fragoso Santos

INESC-ID/Instituto Superior Técnico, Universidade de Lisboa, Portugal
Imperial College London, United Kingdom
jose.fragoso@tecnico.ulisboa.pt

Petar Maksimović

Imperial College London, United Kingdom
p.maksimovic@imperial.ac.uk

Philippa Gardner

Imperial College London, United Kingdom
p.gardner@imperial.ac.uk

Abstract

We introduce a trusted infrastructure for the symbolic analysis of modern event-driven Web applications. This infrastructure consists of reference implementations of the DOM Core Level 1, DOM UI Events, JavaScript Promises and the JavaScript `async/await` APIs, all underpinned by a simple Core Event Semantics which is sufficiently expressive to describe the event models underlying these APIs. Our reference implementations are trustworthy in that three follow the appropriate standards line-by-line and all are thoroughly tested against the official test-suites, passing all the applicable tests. Using the Core Event Semantics and the reference implementations, we develop JaVerT.Click, a symbolic execution tool for JavaScript that, for the first time, supports reasoning about JavaScript programs that use multiple event-related APIs. We demonstrate the viability of JaVerT.Click by proving both the presence and absence of bugs in real-world JavaScript code.

2012 ACM Subject Classification Software and its engineering → Formal software verification; Software and its engineering → Software testing and debugging

Keywords and phrases Events, DOM, JavaScript, promises, symbolic execution, bug-finding

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2020.28

Supplementary Material ECOOP 2020 Artifact Evaluation approved artifact available at <https://doi.org/10.4230/DARTS.6.2.5>.

Funding Fragoso Santos, Gardner, and Maksimović were partially supported by the EPSRC Programme Grant “REMS: Rigorous Engineering for Mainstream Systems” (EP/K008528/1) and the EPSRC Fellowship “VetSpec: Verified Trustworthy Software Specification” (EP/R034567/1).

Gabriela Sampaio: Sampaio was supported by a CAPES Foundation Scholarship, process number 88881.129599/2016-01.

José Fragoso Santos: Fragoso Santos was partially supported by national funds through Fundação para a Ciência e a Tecnologia (FCT), with reference UIDB/50021/2020 (INESC-ID multi-annual funding).



© Gabriela Sampaio, José Fragoso Santos, Petar Maksimović, and Philippa Gardner; licensed under Creative Commons License CC-BY

34th European Conference on Object-Oriented Programming (ECOOP 2020).

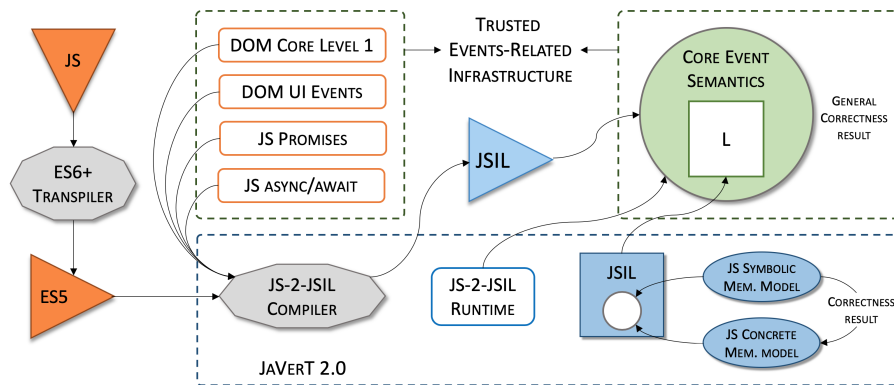
Editors: Robert Hirschfeld and Tobias Pape; Article No. 28; pp. 28:1–28:29

Leibniz International Proceedings in Informatics



LIPICs Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany





■ **Figure 1** Infrastructure of JaVerT.Click.

1 Introduction

Event-driven programming lies at the core of modern Web applications, facilitated by a variety of APIs, such as DOM UI Events [53], JavaScript (JS) Promises [7] and Web Workers [52], each of which comes with its own event model and idiosyncrasies. There has been work on formalising and reasoning about some of these event models: e.g., Rajani et al. [29] have given a formal semantics of DOM UI Events, instrumented to disallow insecure information flows; Lerner et al. [19] have given a formal model and have proven several meta-properties of the DOM Event Dispatch algorithm; and Madsen et al. [22] have developed a calculus for reasoning about JS promises. In each case, the work targets a specific API and its associated event model, and it is not apparent how the work can be extended to include other APIs.

We introduce a trusted infrastructure for the symbolic analysis of modern event-driven Web applications which, we believe for the first time, supports reasoning about code that uses multiple event-related APIs within a single, unified formalism. This infrastructure comprises:

1. a Core Event Semantics, which identifies the fundamental building blocks underpinning the event models of widely-used APIs, and which is formalised and implemented parametrically, assuming an underlying language L (§2); and
2. trusted JS reference implementations of DOM Core Level 1, DOM UI Events, JS promises, and the JS `async/await` (§3-4), the APIs that we target in this paper.

Our infrastructure can readily be added on top of existing symbolic analysis tools; in this paper, we connect it to JaVerT 2.0 [13], a state-of-the-art symbolic analysis tool for JS, creating JaVerT.Click, the first symbolic analysis tool that can reason about JS programs that use multiple event-related APIs. We use JaVerT.Click to analyse `cash` [55] and `p-map` [35], two real-world JS libraries that interact with the targeted APIs, finding bugs in both and establishing bounded correctness of several important properties for `cash`.

The infrastructure of JaVerT.Click is illustrated in Figure 1. JaVerT.Click is built on top of JaVerT 2.0 [13], which supports three types of analysis: whole-program symbolic testing, verification, and automatic compositional testing based on bi-abduction; in this paper, we focus only on symbolic testing. The symbolic execution engine of JaVerT 2.0 works on JSIL, a simple intermediate language that can be instantiated with either the concrete or symbolic memory model of JS. JSIL comes with a correctness result that states that its symbolic testing has no false positives. JaVerT 2.0 targets the strict mode of the ECMAScript 5 standard (ES5 Strict), and comes with: JS-2-JSIL, a trusted compiler from ES5 Strict to JSIL which preserves the memory model and the semantics line-by-line, and is tested using the official Test262 test suite [6]; and the JS-2-JSIL runtime, which provides JSIL implementations of the ES5 Strict internal and built-in functions.

Our reference implementations are all written in ES5 Strict and get compiled to JSIL using JS-2-JSIL as part of JaVerT.Click. These implementations are trusted in that all except that of JS `async/await` (cf. §4.2) follow the API standards line-by-line and all are thoroughly tested against the official test suites, passing all the applicable tests. During the testing, we have discovered coverage gaps in the test suites of DOM Core Level 1 and UI Events and created additional tests to fill these gaps. Our choice to use JS as the API implementation language enables us to directly build on our previous JS analysis, simplifies implementations of promises and `async/await`, which rely on JS for some of their functionality, and makes the implementations easily reusable by other symbolic analysis tools for JS.

As our programs of interest use JS features beyond ES5 Strict, such as `async/await` and anonymous lambda-functions, we introduce a transpilation step from ES6+ Strict to ES5 Strict. This transpiled program and the compiled API reference implementations are then compiled to JSIL using the JS-2-JSIL compiler. The resulting JSIL code, together with the JS-2-JSIL runtime, is passed to the Core Event Semantics instantiated with either the JSIL concrete semantics (for testing) or the JSIL symbolic semantics (for analysis). Assuming correctness of the underlying language (e.g. JSIL), we give a general correctness result for the Core Event Semantics, proving that it has no false positives.

We apply JaVerT.Click to real-world JS code that calls the APIs studied in this paper (§5). In particular, we provide comprehensive symbolic testing of the `events` module of the `cash` library [55], a widely-used alternative for jQuery, which makes heavy use of DOM UI Events. We create a symbolic test suite for the `events` module with 100% line coverage, establishing bounded correctness of several important properties of the module, such as: ‘a handler can be executed for a given event if and only if it has been previously registered for that event’, and also discovering two subtle, previously unknown bugs. We also symbolically test the small, yet widely-used, `p-map` library [35], which uses JS promises and `async/await` to provide an extra layer of functionality on top of JS promises. We achieve 100% line coverage, discovering one bug. All discovered bugs have been reported and have since been fixed.

We believe that our infrastructure can straightforwardly be extended to support other event-driven Web APIs, such as File [51], `postMessage` [54], and Web Workers [52]. This would require a trusted JS reference implementation of the target API and, possibly, an extension of the Core Event Semantics with primitives that handle new types of event behaviour.

2 Core Event Semantics

Our ultimate goal is to develop a formalism within which one could reason symbolically about all event-related APIs. In this paper, we take an important step towards this goal by distilling the essence of three fundamental, complex such APIs – DOM UI events, JS promises, and JS `async/await` – into a minimal Core Event Semantics (onward: Event Semantics) that is easily extensible with support for further APIs. We define the Event Semantics parametrically, as a layer on top of the semantics of a given underlying language (L), thus focussing only on event-related details and filtering out any clutter potentially introduced by the L-semantics. The Event Semantics interacts with the L-semantics by exposing a set of *labels*, which correspond to the fundamental operations underpinning the targeted APIs, such as event handler registration/deregistration and synchronous/asynchronous event dispatch. In this section, we first define the main concepts of the Event Semantics and explain the intuition behind them (§2.1), and then present the concrete (§2.2) and symbolic (§2.3) Event Semantics, connected with an appropriate correctness result.

Values	Event Types	Function Ids	Handler Registers	L-Confs	Conf. Preds
$v \in \mathcal{V}$	$e \in \mathcal{E} \subset \mathcal{V}$	$f \in \mathcal{F} \subset \mathcal{V}$	$h \in \mathcal{H} : \mathcal{E} \rightarrow \overline{\mathcal{F}}$	$c \in \mathcal{C}$	$p \in \mathcal{P} : \mathcal{C} \rightarrow \mathbb{B}$
Event Labels					
$\ell \in \mathcal{L} := \text{addHdlr}\langle e, f \rangle \mid \text{remHdlr}\langle e, f \rangle \mid \text{sDispatch}\langle e, v \rangle \mid \text{aDispatch}\langle e, v \rangle \mid \text{schedule}\langle f, v \rangle \mid \text{await}\langle v, p \rangle$					
Continuations	Continuation Queues	E-Configurations			
$\kappa \in \mathcal{K} := (f, v) \mid (c, p)$	$q \in \mathcal{Q} : \overline{\mathcal{K}}$	$\omega \in \Omega : \mathcal{C} \times \mathcal{H} \times \mathcal{Q}$			

■ **Figure 2** Main Concepts of the Event Semantics.

2.1 Main Concepts of the Event Semantics

The main concepts of our Event Semantics are given in Figure 2. The Event Semantics inherits its *values*, $v \in \mathcal{V}$, from the corresponding L-semantics: for example, if the L-semantics is concrete, these values will be concrete; analogously, if it is symbolic, they will be symbolic. In the meta-theory, we assume that the L-values contain: a distinguished set of unique *event types*, $e \in \mathcal{E}$, intuitively corresponding to, for example, `click` or `focus` in the DOM; and a distinguished set of unique *function identifiers*, $f \in \mathcal{F}$. In the implementation, we represent both as strings. For simplicity, we onward refer to event types as *events*. Our modelling of events is guided by the DOM, in the sense that each event is associated with a list of *handlers*: that is, the functions that should be executed when that event is triggered; this information is kept by the Event Semantics in *handler registers*, $h \in \mathcal{H}$.

The Event Semantics, expectedly, needs to be aware of the configurations of the underlying language (L-configurations), $c \in \mathcal{C}$, but sees them as a black box and interacts with them only through an interface, presented shortly. It does assume that an L-configuration can be divided into: a *store component*, describing the variable store of L; and a *heap component*, describing the heap on which L-execution operates; and a *control flow component*, describing how the L-execution is to proceed. For example, a concrete JSIL configuration, $\langle \rho, \mu, m, cs, i \rangle$, consists of: a variable store ρ (the store component); a memory μ and a metadata table m (the heap component); and a call stack cs for capturing nested function calls and the index of the next command to be executed, i (the control flow component). A symbolic JSIL configuration also includes a path condition, π , which is part of the control flow component. An L-configuration is *final* iff it cannot be executed further in the L-semantics. To model correctly the synchronous dispatch of the DOM and the asynchronous wait of the JS `await`, we also require boolean predicates on L-configurations, $p \in \mathcal{P}$.

The L-semantics communicates with the Event Semantics via *event labels*, $\ell \in \mathcal{L}$, which represent the fundamental operations (primitives) through which we capture the behaviour of our targeted APIs. In particular, `addHdlr` and `remHdlr`, respectively, allow us to add and remove handlers for a given event, whereas `sDispatch` and `aDispatch`, respectively, allow us to dispatch events either *synchronously* (corresponding to the DOM programmatic dispatch) or *asynchronously* (corresponding to a user event, such as clicking a button on a Web page). These four labels are used in the modelling of DOM UI Events (cf. 3.2). Additionally, we support asynchronous computation scheduling via the `schedule` label, required for JS promises (cf. 4.1), and an asynchronous wait via the `await` label, required for JS `await` (cf. 4.2).

All three targeted APIs work with an underlying queue of computations: for the DOM, this queue is implicitly formed by event dispatch; for JavaScript promises and `async/await`, this queue is the job queue of JavaScript. We model these queues as a unified *continuation queue*, $q \in \mathcal{Q}$, which is, essentially, a list of *continuations*, $\kappa \in \mathcal{K}$, which describe how the execution of the Event Semantics is to proceed. We consider two types of continuations:

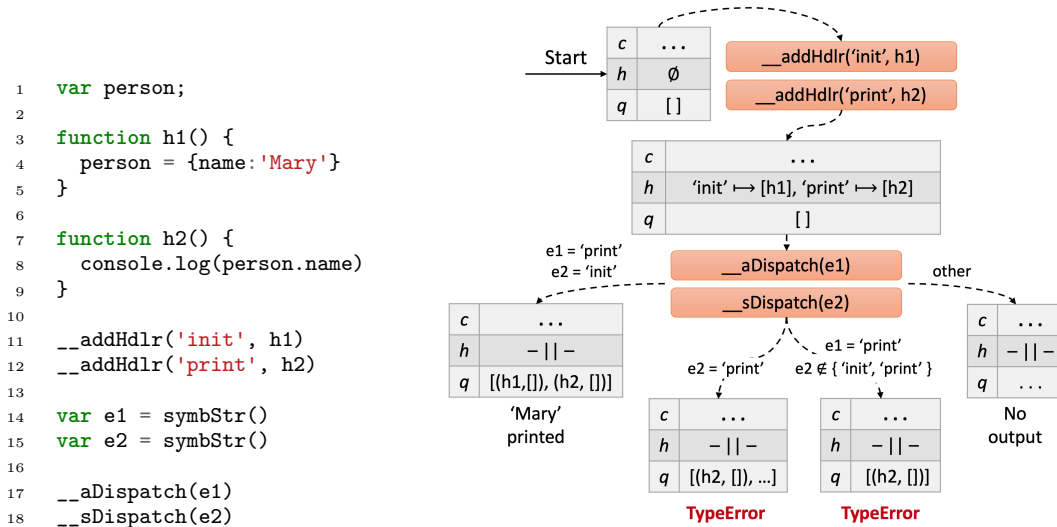
handler-continuations and yield-continuations. A *handler-continuation* is a pair, (f, v) , essentially stating that the handler f is to be executed with argument v . When an event is dispatched via `sDispatch` or `aDispatch`, the respective handler-continuations are put in the handler queue. A *yield-continuation* is a pair, (c, p) , stating that the L-configuration c has been suspended and can be re-activated once the predicate p holds.

Finally, the Event Semantics configurations, (E-configurations), $\omega \in \Omega$, consist of: an L-configuration; a handler register; and a continuation queue.

Using the Event Semantics in JavaScript. Our JS reference implementations of the event-related APIs interact with the Event Semantics via *JS wrapper functions*, one per event label; we denote, for example, the wrapper function of the `addHdlr` label by `__addHdlr`, and the others analogously. Calls to these wrapper functions are meant to be intercepted by the underlying JavaScript implementation, which is then supposed to construct the corresponding label and pass it on to the Event Semantics. In `JaVerT.Click`, these wrapper functions resolve to JSIL functions with dedicated identifiers, the calls to which are then intercepted appropriately by the JSIL semantics. This approach, however, is independent of `JaVerT.Click`: any other implementation of JavaScript and of our Event Semantics can re-use our reference implementations, as long as these wrapper functions are properly intercepted.

Example. Below, we give a simple JavaScript example of how our Event Semantics can be used in `JaVerT.Click` (left), together with parts of its execution trace (right). In the E-configurations shown in the trace, we focus on the handler register and continuation queue, both of which are initially empty, and omit the details of the JSIL-configuration c .

First, in lines 1-9, we declare a variable `person` and two functions: `h1`, which initialises `person`, and `h2`, which prints out its `name`. Next, in lines 11-12, we add `h1` and `h2` as handlers for the `'init'` and `'print'` events, respectively, by using the wrapper function `__addHdlr`, exposed globally by the Event Semantics. This is recorded appropriately in the handler register, which then does not change for the remainder of the execution (denoted by `-||-` in the diagram). Next, in lines 14-15, we declare `e1` and `e2` to be two symbolic events (strings), using the `symbStr()` function of `JaVerT.Click`. Finally, we dispatch `e1` asynchronously (line 17) and `e2` synchronously (line 18), using the appropriate wrapper functions. Intuitively,



in an asynchronous dispatch, the related handlers (here, any handlers for e_1) are added to the *back* of the current continuation queue (here, an empty continuation queue), to be executed after all of the previously scheduled continuations are completed. In contrast, in a synchronous dispatch, the current computation is suspended and the related handlers (here, any handlers for e_2) are added to the *front* of the continuation queue (which now contains the handlers for e_1), to be executed immediately, followed by the remainder of the suspended current computation (which is empty, as the synchronous dispatch is at the end of the program, and is thus omitted from the diagram).

Given that the events are symbolic, the two dispatches will cause the execution of `JaVerT.Click to branch`; there are four relevant cases, as illustrated in the diagram. First, if e_1 equals `'print'` and e_2 equals `'init'`, the continuation queue after the dispatches will contain h_1 followed by h_2 , meaning that the execution will terminate successfully and `'Mary'` will be printed to the console. However, if e_2 equals `'print'` (meaning that h_2 will be put in the front of the continuation queue by the synchronous dispatch) or if e_1 equals `'print'` and e_2 has no associated handlers (meaning that h_2 will be put in the back of the continuation queue by the asynchronous dispatch, but will be the only function in that queue), the execution will throw a native JavaScript type error, as h_2 will attempt to read the `'name'` property of `person`, which will not have been initialised. Finally, in all other cases, the execution will terminate successfully, but with no output to the console.

Parametricity of the Event Semantics. As illustrated in Figure 1, the Event Semantics is implemented parametrically, as a layer on top of a given L-semantics. Since a unified presentation that reflects the implementation precisely would take up considerable space, we choose to present the concrete (§2.2) and the symbolic (§2.3) Event Semantics separately.

2.2 Concrete Event Semantics

A concrete Event Semantics is built on top of a concrete L-semantics. It interacts with L-configurations via an interface that consists of six functions: `assume`, `suspend`, `initialConf`, `isFinal`, `mergeConfs`, and `splitReturn`; we describe these functions abstractly on their first use, and illustrate how some of them work in JSIL. The Event Semantics also uses the following auxiliary relations: (1) *add handler*, $\mathcal{AH}(h, e, f)$, for extending the handler register h with the handler f for an event e ; (2) *remove handler*, $\mathcal{RH}(h, e, f)$, for removing the handler f for e from h ; (3) *find handlers*, $\mathcal{FH}(h, e)$, for obtaining the handlers associated with e in h ; and (4) *continue with*, $\mathcal{CW}_L(c, \kappa)$, for updating the L-configuration c so that the continuation κ can be executed. We first give the formal definitions of these auxiliary relations, using function notation as they are deterministic in the concrete case. We write $\#$ to denote list concatenation; $h_o(e)$ to denote $h(e)$ if it is defined, and the empty list otherwise; and $l \setminus f$ to denote the list obtained from the list l by removing all occurrences of f .

Concrete Event Semantics: Auxiliary Relations

ADD HANDLER $\mathcal{AH}(h, e, f) \triangleq$ $h[e \mapsto h_o(e) \# [f]]$	FIND HANDLER $\mathcal{FH}(h, e) \triangleq h_o(e)$	CW-HANDLER-CONT. $\mathcal{CW}_L(c, (f, v)) \triangleq L.\text{initialConf}(c, (f, v))$
REMOVE HANDLER $\mathcal{RH}(h, e, f) \triangleq$ $\begin{cases} h[e \mapsto h(e) \setminus f], & \text{if } e \in \text{dom}(h) \\ h, & \text{otherwise} \end{cases}$	CW-YIELD-CONT. $\frac{p(c) = \text{True}}{\mathcal{CW}_L(c, (c', p)) \triangleq L.\text{mergeConfs}(c, c')}$	

These definitions are all straightforward except \mathcal{CW}_L . When given a handler-continuation, $\kappa = (f, v)$, \mathcal{CW}_L sets up the execution of the handler f with argument v by using the `initialConf` function of the L-semantic interface, which returns an the L-configuration consisting of the the heap component of c and the control flow and store components set up to execute only the function f with argument v . When given a yield-continuation, $\kappa = (c', p)$, \mathcal{CW}_L requires the predicate p to hold for the current L-configuration c , in which case it merges the two configurations using the `mergeConfs`(c, c') function of the L-semantic interface, which returns a configuration that consists of the heap component of c and the control flow and store components of c' ; in particular, in JSIL, given $c = \langle \rho, \mu, m, cs, i \rangle$ and $c' = \langle \rho', \mu', m', cs', i' \rangle$, we would have that `mergeConfs`(c, c') = $\langle \rho', \mu, m, cs', i' \rangle$.

We now give the concrete Event Semantics transitions, which are of the form $\omega \xrightarrow{\alpha}_{\mathbf{E}(\mathbf{L})} \omega'$, where ω and ω' , respectively, are the configurations before and after the computed step, and α is an environment action. Environment actions are used to model events triggered by the environment, such as user UI-events and network events. They have the grammar $\alpha ::= \cdot \mid (e, v)$, where \cdot represents no environment action and (e, v) represents the triggering of the event e with value v . For clarity, we elide \cdot in the transitions.

Concrete Event Semantics: $\langle c, h, q \rangle \xrightarrow{\alpha}_{\mathbf{E}(\mathbf{L})} \langle c', h', q' \rangle$

LANGUAGE TRANSITION $c \xrightarrow{\ell}_{\mathbf{L}} c'$	ADD HANDLER $c \xrightarrow{\ell}_{\mathbf{L}} c' \quad \ell = \text{addHdlr}\langle e, f \rangle$	REMOVE HANDLER $c \xrightarrow{\ell}_{\mathbf{L}} c' \quad \ell = \text{remHdlr}\langle e, f \rangle$
$\langle c, h, q \rangle \xrightarrow{\mathbf{E}(\mathbf{L})} \langle c', h, q \rangle$	$\langle c, h, q \rangle \xrightarrow{\mathbf{E}(\mathbf{L})} \langle c', \mathcal{AH}(h, e, f), q \rangle$	$\langle c, h, q \rangle \xrightarrow{\mathbf{E}(\mathbf{L})} \langle c', \mathcal{RH}(h, e, f), q \rangle$
SYNCHRONOUS DISPATCH $c \xrightarrow{\ell}_{\mathbf{L}} c' \quad \ell = \text{sDispatch}\langle e, v \rangle \quad [f_i \mid_0^n] = \mathcal{FH}(h, e)$ $q' = [(f_i, [e, v]) \mid_{i=0}^n] \quad c'' = \text{L.suspend}(c')$		ASYNCHRONOUS DISPATCH $c \xrightarrow{\ell}_{\mathbf{L}} c' \quad \ell = \text{aDispatch}\langle e, v \rangle$ $[f_i \mid_0^n] = \mathcal{FH}(h, e) \quad q' = [(f_i, [e, v]) \mid_{i=0}^n]$
$\langle c, h, q \rangle \xrightarrow{\mathbf{E}(\mathbf{L})} \langle c'', h, q' \# [(c', (\lambda c. \text{True}))] \# q \rangle$		$\langle c, h, q \rangle \xrightarrow{\mathbf{E}(\mathbf{L})} \langle c', h, q \# q' \rangle$
SCHEDULE $c \xrightarrow{\ell}_{\mathbf{L}} c' \quad \ell = \text{schedule}\langle f, v \rangle$ $q' = q \# [(f, v)]$	AWAIT $c \xrightarrow{\ell}_{\mathbf{L}} c' \quad \ell = \text{await}\langle v, p \rangle$ $(c_r, c_a) = \text{L.splitReturn}(c', v)$	ENVIRONMENT DISPATCH $[f_i \mid_0^n] = \mathcal{FH}(h, e)$ $q' = [(f_i, [e, v]) \mid_{i=0}^n]$
$\langle c, h, q \rangle \xrightarrow{\mathbf{E}(\mathbf{L})} \langle c', h, q' \rangle$	$\langle c, h, q \rangle \xrightarrow{\mathbf{E}(\mathbf{L})} \langle c_r, h, q \# [(c_a, p)] \rangle$	$\langle c, h, q \rangle \xrightarrow{\mathbf{E}(\mathbf{L})}^{(e, v)} \langle c, h, q \# q' \rangle$
CONTINUATION-SUCCESS $\text{L.isFinal}(c) \quad q = \kappa : q'$		CONTINUATION-FAILURE $\text{L.isFinal}(c) \quad q = \kappa : q' \quad (c, \kappa) \notin \text{dom}(\mathcal{CW}_L)$
$\langle c, h, q \rangle \xrightarrow{\mathbf{E}(\mathbf{L})} \langle \mathcal{CW}_L(c, \kappa), h, q' \rangle$		$\langle c, h, q \rangle \xrightarrow{\mathbf{E}(\mathbf{L})} \langle c, h, q' \# [\kappa] \rangle$

The first seven rules rely on a transition of the L-semantic, updating the current L-configuration with the one generated by the L-transition and using the generated label to determine which event-related action is to be performed, if any. The first three rules are straightforward; we describe the remaining four below:

[Synchronous Dispatch] When the L-semantic generates the label `sDispatch`(e, v), the Event Semantics first creates a handler-continuation for each handler associated with e , together with a yield continuation, $(c', (\lambda c. \text{True}))$. These continuations are then all added to the *front* of the continuation queue, ensuring that the handlers will be executed in order, after which the current computation will be retaken unconditionally, given [CW-YIELD-CONT.]. Lastly, the Event Semantics uses the `suspend`(c') function of the L-semantic, which returns the configuration that is the same as c' but marked as final, to construct a final configuration c'' , which, given [CONTINUATION-SUCCESS], means that the execution of c' will stop and the first handler will be executed next.

[Asynchronous Dispatch] When the L-semantic generates the label $\text{aDispatch}\langle e, v \rangle$, the Event Semantics proceeds similarly to [SYNCHRONOUS DISPATCH], but the continuations are added to the *back* of the continuation queue rather than to the front, meaning that the handlers will still be executed in order, but at some point in the future.

[Schedule] The L-semantic generates the label $\text{schedule}\langle f, v \rangle$; the Event Semantics creates a handler-continuation (f, v) for the given function with the given arguments and places it at the *back* of the continuation queue.

[Await] When the L-semantic generates the label $\text{await}\langle v, p \rangle$, the Event Semantics creates the return configuration, c_r , and the await configuration, c_a via the `splitReturn` function of the L-semantic interface, which constructs: c_r from c by setting up the control flow component as if the currently executing function, f , returned the value v ; and c_a from c by setting up the control flow component to only contain the remainder of the execution of f . It then schedules the remainder of the computation of the currently executing function to be completed asynchronously once p holds, and continues the current computation as if the currently executing function had returned the value v .

The remaining three transitions do not rely on the L-semantic. In the [ENVIRONMENT DISPATCH] case, the environment generates the label (e, v) , and the Event Semantics behaves as for [ASYNCHRONOUS DISPATCH], except that the resulting L-configuration does not change. If the current active configuration is final (as checked by the `isFinal(c)` function of the L-semantic interface, which returns true if c is final, and false otherwise), the Event Semantics tries to create a new configuration for the execution of the continuation at the front of the continuation queue. If this is possible, the execution proceeds ([CONTINUATION-SUCCESS]); otherwise, that continuation is demoted to the back of the continuation queue ([CONTINUATION-FAILURE]).

2.3 Symbolic Event Semantics

Symbolic execution [2, 3, 4] is a program analysis technique that systematically explores all possible executions of the given program up to a bound, by executing the program on symbolic values instead of concrete ones. For each execution path, symbolic execution constructs a first-order quantifier-free formula, called a *path condition*, which accumulates the constraints on the symbolic inputs that direct the execution along that path. Here, we describe a symbolic version of the Event Semantics introduced in §2.2, obtained by lifting the concrete event semantics to the symbolic level, following well-established approaches [44, 43, 12].

We assume that L has a symbolic semantics with symbolic values, $\hat{v} \in \hat{\mathcal{V}}$, built using symbolic variables, $\hat{x} \in \hat{\mathcal{X}}$. The concepts introduced in §2.1 are defined as in Figure 2, but for symbolic instead of concrete values, and are annotated with $\hat{\cdot}$ to be distinguishable from their concrete counterparts; e.g., we have: *symbolic events*, $\hat{e} \in \hat{\mathcal{E}} \subset \hat{\mathcal{V}}$; *symbolic handler registers*, $\hat{h} \in \hat{\mathcal{H}}$, mapping symbolic events to lists of function identifiers; and symbolic configurations, $\hat{\omega} \in \hat{\mathcal{Q}}$, comprising a symbolic L-configuration, $\hat{c} \in \hat{\mathcal{C}}f$, a *symbolic handler register*, and a *symbolic continuation queue*, $\hat{q} \in \hat{\mathcal{Q}}$. We also assume that every symbolic L-configuration \hat{c} contains a boolean symbolic value, $\pi \in \Pi \subset \hat{\mathcal{V}}$, to which we refer as the *path condition* of \hat{c} .

The symbolic Event Semantics, like the concrete, uses the L-semantic interface and the four auxiliary relations introduced in §2.2. When executed symbolically, however, the auxiliary relations that operate on handler registers (\mathcal{AH} , \mathcal{RH} , and \mathcal{FH}) may branch. To account for this branching, we pair each outcome with a constraint describing the conditions under which the outcome is valid. The formal definitions are given below; we omit the definition of the \mathcal{RH} relation, as it is analogous to that of \mathcal{AH} .

Symbolic Event Semantics: Auxiliary Relations

$\frac{\text{ADD HANDLER - FOUND} \quad \hat{e}' \in \text{dom}(\hat{h}) \quad \hat{h}' = \hat{h} [\hat{e}' \mapsto \hat{h}(\hat{e}') + [f]]}{\mathcal{AH}(\hat{h}, \hat{e}, f) \rightsquigarrow (\hat{h}', \hat{e} = \hat{e}')}$	$\frac{\text{ADD HANDLER - NOT FOUND} \quad \hat{h}' = \hat{h} [\hat{e} \mapsto [f]]}{\mathcal{AH}(\hat{h}, \hat{e}, f) \rightsquigarrow (\hat{h}', \hat{e} \notin \text{dom}(\hat{h}))}$
$\frac{\text{FIND HANDLER - FOUND} \quad \hat{e}' \in \text{dom}(\hat{h})}{\mathcal{FH}(\hat{h}, \hat{e}) \rightsquigarrow (\hat{h}(\hat{e}'), \hat{e} = \hat{e}')}$	$\frac{\text{FIND HANDLER - NOT FOUND}}{\mathcal{FH}(\hat{h}, \hat{e}) \rightsquigarrow ([], \hat{e} \notin \text{dom}(\hat{h}))}$

An excerpt of the symbolic Event Semantics is given below. We focus on the representative rules different from their concrete counterparts, highlighting the differences in grey. These differences are introduced by the above-discussed branching of the auxiliary relations; in particular, every time an auxiliary relation is used, the constraint it generates must be added to the current path condition using the `assume`(\hat{c}, π) function of the L-semantic interface, which returns the symbolic L-configuration obtained by extending the path condition of \hat{c} with the formula π if such an extension is satisfiable, and is undefined otherwise.

Symbolic Event Semantics (excerpt): $\langle \hat{c}, \hat{h}, \hat{q} \rangle \rightsquigarrow_{\hat{\mathbf{E}}(\mathbf{L})}^{\hat{\mathbf{c}}} \langle \hat{c}', \hat{h}', \hat{q}' \rangle$

$\frac{\text{ADD HANDLER} \quad \hat{c} \rightsquigarrow_{\mathbf{L}}^{\hat{\mathbf{c}}} \hat{c}' \quad \hat{\ell} = \text{addHdlr}(\hat{e}, f)}{\mathcal{AH}(\hat{h}, \hat{e}, f) \rightsquigarrow (\hat{h}', \pi) \quad \hat{c}'' = \text{L.assume}(\hat{c}', \pi)} \quad \langle \hat{c}, \hat{h}, \hat{q} \rangle \rightsquigarrow_{\hat{\mathbf{E}}(\mathbf{L})} \langle \hat{c}'', \hat{h}', \hat{q} \rangle$	$\frac{\text{ENVIRONMENT DISPATCH} \quad \mathcal{FH}(\hat{h}, \hat{e}) \rightsquigarrow ([f_i \mid_0^n], \pi) \quad \hat{q}' = [(f_i, [\hat{e}, \hat{v}]) \mid_{i=0}^n]}{\hat{c}' = \text{L.assume}(\hat{c}, \pi)} \quad \langle \hat{c}, \hat{h}, \hat{q} \rangle \rightsquigarrow_{\hat{\mathbf{E}}(\mathbf{L})}^{(\hat{e}, \hat{v})} \langle \hat{c}', \hat{h}, \hat{q} + \hat{q}' \rangle$
$\frac{\text{SYNCHRONOUS DISPATCH} \quad \hat{c} \rightsquigarrow_{\mathbf{L}}^{\hat{\mathbf{c}}} \hat{c}' \quad \hat{\ell} = \text{sDispatch}(\hat{e}, \hat{v})}{\mathcal{FH}(\hat{h}, \hat{e}) \rightsquigarrow ([f_i \mid_0^n], \pi) \quad \hat{q}' = [(f_i, [\hat{e}, \hat{v}]) \mid_{i=0}^n]} \quad \hat{c}'' = \text{L.assume}(\hat{c}', \pi) \quad \hat{c}''' = \text{L.suspend}(\hat{c}'') \quad \langle \hat{c}, \hat{h}, \hat{q} \rangle \rightsquigarrow_{\hat{\mathbf{E}}(\mathbf{L})} \langle \hat{c}''', \hat{h}, \hat{q}' + [(\hat{c}'', (\lambda \hat{c}. \text{True}))] + \hat{q} \rangle$	$\frac{\text{ASYNCHRONOUS DISPATCH} \quad \hat{c} \rightsquigarrow_{\mathbf{L}}^{\hat{\mathbf{c}}} \hat{c}' \quad \hat{\ell} = \text{aDispatch}(\hat{e}, \hat{v})}{\mathcal{FH}(\hat{h}, \hat{e}) \rightsquigarrow ([f_i \mid_0^n], \pi) \quad \hat{q}' = [(f_i, [\hat{e}, \hat{v}]) \mid_{i=0}^n]} \quad \hat{c}'' = \text{L.assume}(\hat{c}', \pi) \quad \langle \hat{c}, \hat{h}, \hat{q} \rangle \rightsquigarrow_{\hat{\mathbf{E}}(\mathbf{L})} \langle \hat{c}'', \hat{h}, \hat{q} + \hat{q}' \rangle$

Correctness. To establish the correctness of the symbolic Event Semantics w.r.t the concrete Event Semantics, we first relate the corresponding configurations using *symbolic environments*, $\varepsilon : \hat{\mathcal{X}} \rightarrow \mathcal{V}$, which map symbolic variables to concrete values, while preserving types. Given a symbolic environment ε , we write $\mathcal{I}_\varepsilon(\hat{v})$ to denote the interpretation of \hat{v} under ε , with the key case being that of symbolic variables: $\mathcal{I}_\varepsilon(\hat{x}) = \varepsilon(\hat{x})$. We extend \mathcal{I}_ε to all other concepts defined in Figure 2 component-wise, overloading notation: for example, $\mathcal{I}_\varepsilon(\langle \hat{c}, \hat{h}, \hat{q} \rangle) \triangleq \langle \mathcal{I}_\varepsilon(\hat{c}), \mathcal{I}_\varepsilon(\hat{h}), \mathcal{I}_\varepsilon(\hat{q}) \rangle$. We assume that interpretation is preserved by the functions of the L-semantic interface; for example, that $\text{L.isFinal}(\hat{c}) \Leftrightarrow \text{L.isFinal}(\mathcal{I}_\varepsilon(\hat{c}))$.

We define the *models* of a symbolic L-configuration \hat{c} under the path condition π as the set of all concrete configurations obtained via interpretations of \hat{c} that satisfy π and their accompanying symbolic environments: $\mathcal{M}_\pi(\hat{c}) = \{(\varepsilon, \mathcal{I}_\varepsilon(\hat{c})) \mid \mathcal{I}_\varepsilon(\pi) = \text{True}\}$. We extend this notion to symbolic labels, environment actions, and E-configurations, overloading notation.

The correctness of the Event Semantics relies on the correctness of the L-semantic. A given symbolic L-semantic is correct w.r.t. a given concrete L-semantic, as formalised in Definition 1, if every symbolic trace: **(1)** over-approximates all concrete traces that follow its execution path and whose initial concrete L-configuration is over-approximated by the initial symbolic L-configuration (*Directed Soundness*); and **(2)** has at least one concretisation (*Directed Completeness*). Directed Completeness, in particular, guarantees the absence of false-positive bug-reports: if a bug happens symbolically, then it must also happen concretely.

► **Definition 1** (Correctness Criteria - Symbolic L-Semantics).

$$\begin{array}{ll}
 \text{\textit{L-DIRECTED-SOUNDNESS}} & \text{\textit{L-DIRECTED-COMPLETENESS}} \\
 \hat{c} \rightsquigarrow_{\mathbb{L}}^{\hat{\ell}} c' \wedge (\pi \Rightarrow \text{pc}(\hat{c}')) \wedge (\varepsilon, c) \in \mathcal{M}_{\pi}(\hat{c}) \wedge c \rightsquigarrow_{\mathbb{L}}^{\ell} c' & \hat{c} \rightsquigarrow_{\mathbb{L}}^{\hat{\ell}} c' \wedge (\pi \Rightarrow \text{pc}(\hat{c}')) \\
 \implies (\varepsilon, c') \in \mathcal{M}_{\pi}(\hat{c}') \wedge (\varepsilon, \ell) \in \mathcal{M}_{\pi}(\hat{\ell}) & \wedge (\varepsilon, c) \in \mathcal{M}_{\pi}(\hat{c}) \\
 & \implies \exists \ell, c'. c \rightsquigarrow_{\mathbb{L}}^{\ell} c'
 \end{array}$$

Theorem 2 states that if the symbolic L-semantics is correct, then so is the obtained Event Semantics. To precisely identify the concrete traces that follow the same path as the symbolic trace, in Theorem 2 we only pick concretisations of the initial symbolic state that satisfy the *final* path condition ($\pi = \text{pc}(\hat{\omega}')$).

► **Theorem 2** (Correctness of the Symbolic Event Semantics).

$$\begin{array}{ll}
 \text{\textit{E-DIRECTED-SOUNDNESS}} & \text{\textit{E-DIRECTED-COMPLETENESS}} \\
 \hat{\omega} \rightsquigarrow_{\hat{E}(\mathbb{L})}^{\hat{\alpha}} \hat{\omega}' \wedge \pi = \text{pc}(\hat{\omega}') \wedge (\varepsilon, \omega) \in \mathcal{M}_{\pi}(\hat{\omega}) & \hat{\omega} \rightsquigarrow_{\hat{E}(\mathbb{L})}^{\hat{\alpha}} \hat{\omega}' \wedge \pi = \text{pc}(\hat{\omega}') \\
 \wedge (\varepsilon, \alpha) \in \mathcal{M}_{\pi}(\hat{\alpha}) \wedge \omega \rightsquigarrow_{\hat{E}(\mathbb{L})}^{\alpha} \omega' & \wedge (\varepsilon, \omega) \in \mathcal{M}_{\pi}(\hat{\omega}) \\
 \implies (\varepsilon, \omega') \in \mathcal{M}_{\pi}(\hat{\omega}') & \implies \exists \alpha, \omega'. \omega \rightsquigarrow_{\hat{E}(\mathbb{L})}^{\alpha} \omega'
 \end{array}$$

We actually prove a stronger result, analogous to that given in Definition 1, with $\pi \Rightarrow \text{pc}(\hat{\omega}')$, from which the presented result trivially follows. The proof is done by case analysis on the symbolic rules for the Event Semantics, and can be found integrally in [31].

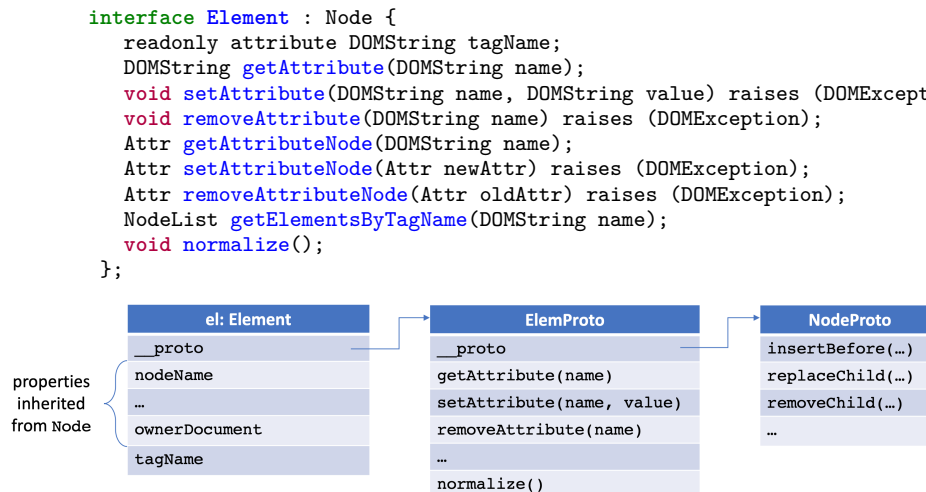
3 The DOM API

The Document Object Model (DOM) [53] is an API through which the code executing in the browser can interact with the Web page displayed to the user. Initially designed as a simple XML/HTML inspect-update library, the DOM has been substantially extended over the last twenty years and now includes a wide variety of features, such as specialised traversals, events, abstract views, and cascading style sheets. To cope with this growing complexity, the DOM API has been organised as a collection of smaller APIs, each targeting a specific set of features. Recently, the most relevant of these APIs, Core Levels 1-3 [47, 49], have been unified in a single all-encompassing DOM API, called the DOM Living Standard [53], which defines a “platform-neutral model for events, aborting activities, and node-trees”. The DOM Living Standard is inspired by the ECMAScript standard [7]. It is written as if it were the pseudo-code of a DOM implementation, describing each DOM method *operationally* and detailing each evaluation step. This approach, unlike the previous declarative one [47, 48, 49], facilitates new reference implementations tightly connected to the text of the standard.

In this section, we present our JavaScript reference implementations of two DOM APIs: DOM Core Level 1 [47], which describes a range of operations for inspecting and updating XML/HTML documents (§3.1); and DOM UI Events [53], which describes the event model of the DOM (§3.2). For the latter, we describe in detail its connection to the Event Semantics. Importantly, both reference implementations are *trustworthy*: they closely follow the specifications of their corresponding methods as per the DOM Living Standard, as illustrated in this section; and they were thoroughly tested against the appropriate official test suites, as shown in §5. They, therefore, constitute a reliable representation of the DOM, which is useful for analysing Web programs that interact with the DOM API.

3.1 DOM Core Level 1

The DOM Core Level 1 API [47] is the first version of the DOM API. It describes how XML/HTML documents are internally represented as DOM trees and defines a range of methods for manipulating these trees. DOM trees comprise several different types of DOM



■ **Figure 3** DOM Element interface (top) and the respective JavaScript object graph (bottom).

nodes and are subject to a number of topological constraints restricting the ways in which these nodes can form a valid DOM tree. For instance, the root node of every DOM tree must have type `Document` and can have at most one child of type `Element`. Elements, on the other hand, can have multiple child nodes of different types, such as `Text` and `Element`.

The DOM standard defines interfaces describing the structure of every type of DOM node in an object-oriented style. For every node type, the standard specifies the fields and methods exposed by the nodes of that type. Furthermore, as in standard OO languages, each node type might *inherit* from another node type; for instance, every `Element` node is also a `Node`, meaning that it exposes all fields and methods defined in the `Node` interface.

We implement the DOM Core Level 1 API in JavaScript (ES5 Strict), encoding DOM objects as JS objects. In particular, each type of DOM node is mapped to the JS constructor function in charge of creating the nodes of that type. Also, we emulate class-based inheritance, which is used to describe DOM nodes in the standard, using the prototype inheritance of JS, by storing the methods shared by all nodes of a given type in their (shared) prototype.

In the following, we describe our implementations of the `Element` and `NodeList` interfaces, which showcase, respectively, how our implementation follows the standard, and how JavaScript enables us to write an elegant implementation of DOM live collections.

Element Interface. In Figure 3, we show the `Element` interface written in IDL (Interface Description Language) as in the standard (top) and a fragment of its corresponding object graph from our JavaScript implementation (bottom). The standard states that `Element` inherits from `Node`, meaning that all objects of type `Element` expose the methods and fields of `Node` objects. Additionally, every `Element` object exposes a field `tagName` and the methods `getAttribute`, `setAttribute`, `removeAttribute`, `getAttributeNode`, `setAttributeNode`, `removeAttributeNode`, `getElementsByTagName`, and `normalize`.

In the JavaScript object graph, besides exposing the property `tagName`, all `Element` objects directly define the properties corresponding to the fields of the `Node` interface (e.g. `nodeName`, `ownerDocument`, etc). The methods of the `Element` interface are stored in the object `ElemProto`, the prototype of all `Element` objects, and the `Node` methods are stored in `NodeProto`, which is the prototype of `ElemProto`.

NodeList Interface. The `NodeList` interface describes the so-called DOM *live collections*. A live collection is a special data structure defined in the DOM API that automatically reflects changes that occur in its associated document. For instance, the `getElementsByTagName` method from the above-mentioned `Element` interface returns a live collection containing the DOM nodes that match the supplied tag name. Working with live collections is error-prone and requires particular attention. Consider, for example, the following program:

```
var divs = body.getElementsByTagName("div");
for (var i = 0; i < divs.length; i++)
  { body.appendChild(document.createElement("div")) }
```

This program iterates over the initial collection of `div` nodes in the DOM tree rooted at `body`. On each iteration, it creates a new `div` node and inserts it into the original tree. However, this new `div` is also inserted into the live collection `divs`, whose length automatically increases by one, causing the program to loop forever.

The `NodeList` interface defines the field `length`, for obtaining the length of a node list, and the method `item(i)` for accessing its i -th element. In JavaScript, we implement node lists *lazily* in that we recompute the contents of a given node list every time it is inspected. This we achieve by extending `NodeList` objects with an internal `compute` function, used to compute its contents. We call `compute` at every invocation of the `item` method, and associate the `length` property of every node list with a JavaScript *getter* that also calls `compute` before checking the length of the corresponding node list. As an optimisation, we cache computed live collections by associating each node list with a unique identifier and maintaining a global array of computed node lists. However, whenever there is any update to the DOM tree, all cached live collections are invalidated and will be re-computed the next time they are inspected.

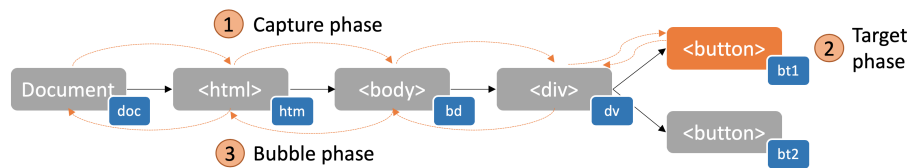
3.2 DOM UI Events

The DOM UI Events API [53] describes the DOM event model. In particular, it provides the mechanism for programmers to register event listeners, and explains how these listeners are collected and executed every time a DOM event gets triggered either by the environment (for example, via user events and browser events) or programatically.

At the core of the UI Events API is the DOM `Dispatch` algorithm, which precisely describes the process of collecting and executing event listeners every time a DOM event gets triggered. The DOM Living standard includes the pseudo-code of the `Dispatch` algorithm, detailing all the steps that are performed when dispatching a DOM event ([53], §2.9). It is a complex algorithm that relies on a number of auxiliary functions, which, in turn, are also described operationally and often rely on other auxiliary functions themselves.

In the following, we describe our implementation of the DOM `Dispatch`, demonstrate that this implementation follows the pseudo-code of the standard line-by-line, and describe in detail how it is connected to the Event Semantics.

DOM Dispatch. We explain the DOM `Dispatch` algorithm via an example given in Figure 4, which shows a DOM tree of an HTML page with an element `dv` containing two buttons, `bt1` and `bt2`, and illustrates the steps taken by `Dispatch` when the user clicks on `bt1`. Coarsely, `Dispatch` first determines the *propagation path* of the triggered event, i.e. the list of DOM nodes connecting the element on which the event was triggered to the root of the DOM document, in this case [`bt1`, `dv`, `bd`, `htm`, `doc`]. Then, it executes the handlers registered along that propagation path during three consecutive phases: (1) the *capture phase*, where the



■ **Figure 4** DOM Dispatch Phases.

event is propagated from the root of the document, `doc`, to the target, `bt1`; (2) the *target phase*, where the event is processed at the target, `bt1`; and (3) the *bubble phase*, where the event is propagated back to the root. During each phase, `Dispatch` executes the handlers attached to the current node if they were registered for the current event and phase. The DOM API method for registering handlers, `addEventListener(type, handler, useCapture)`, allows the programmer to specify if a given handler is to be executed in the capture phase or the bubble phase through the `useCapture` boolean; by default, handlers get executed in the target phase. Importantly, the propagation path is computed only once, before the handlers are executed, meaning that even if their execution alter the propagation path, those changes will not be taken into account by the `Dispatch` algorithm.

Below, we present our JavaScript (ES5 Strict) implementation of the `Dispatch` algorithm. In the standard, `Dispatch` is presented as a monolithic 56-line function that is difficult to understand. We instead structure it into seven auxiliary functions, each following the corresponding pseudo-code of the standard line-by-line.

```

1  function Dispatch(event, target, flags) {
2    var relatedTarget = retarget(event.relatedTarget, target);
3    var touchTargets = getTouchTargets(event, target);
4    var actTarget = isActivationTarget(event);
5    updatePropagationPath(event, target, relatedTarget, touchTargets, actTarget);
6    captureAndTarget(event, flags)
7    if (event.bubbles) { bubble(event, flags) }
8    clear(event);
9    return !event.canceled
10 }

```

The `Dispatch` algorithm receives as input: the `Event` object that represents the triggered event; the `Node` object on which the event was triggered; and optional flags used to identify a target/event requiring special treatment. The algorithm then proceeds as follows:

1. Call `retarget` to determine the *related target* of the triggered event. Some events are associated with two targets: the main target, supplied as the argument of `Dispatch`; and the related target, determined by `retarget`. For instance, `mouseout`, an event triggered when the user moves the mouse from one node to another, has two targets: the node at which the mouse originally was (main), and the node to which it moved (related).
2. Call `getTouchTargets` to obtain the list of *touch targets* associated with the triggered event. Events involving interactions between the user and a touching surface can be associated with a variable number of targets (e.g., due to the user placing multiple fingers on the surface), called touch targets.
3. Call `isActivationTarget` to check if the event has an associated activation behaviour. For instance, when a `click` event is triggered on a hyperlink, the browser should open a window with the corresponding URL.
4. Call `updatePropagationPath` to determine the propagation path of the event.
5. Call `captureAndTarget` to execute the capture and target phases.

6. Call `bubble` to execute the bubble phase if the result of inspecting the property bubbles of the event object is true.
7. Call `clear` to reset some of the properties of the event object to `null`.
8. Return a boolean indicating if the activation behaviour of the event was not cancelled. When no activation behaviour is defined, the algorithm returns `true`.

Using the Event Semantics. In related works [19, 29], the DOM `Dispatch` is either baked into the formalism, which then becomes complex, and/or not fully faithful to the standard. We take a novel, substantially different approach that allows us both to keep the Event Semantics simple and to represent rigorously all of the details of the DOM `Dispatch`. In particular, we store information about DOM handlers directly in their associated `Element` nodes in the JavaScript heap, implement the `Dispatch` fully in JavaScript, and only use the Event Semantics to: (1) register the `Dispatch` function as the handler of *all* DOM events using the `addHdlr` primitive; and (2) dispatch programmatic DOM events synchronously using the `sDispatch` primitive. The former effectively means that any time a DOM event (e.g. `click` or `focus`), is triggered, either synchronously or asynchronously, the DOM `Dispatch` function itself is scheduled for execution by the Event Semantics. It is then the job of this function, rather than the Event Semantics, to traverse the DOM tree, starting at the node where the event was triggered, and execute the user-register handlers in the appropriate order.

Below, we show our implementation of the `dispatchEvent` function, used to model programmatic dispatch of DOM events. This function calls the Event Semantics synchronous dispatch wrapper, `__sDispatch`, in line 5. The behaviour of the `sDispatch` primitive, as given in §2, precisely captures the programmatic DOM event dispatch as per the standard, where the associated event handlers are meant to be executed immediately.

```

1 function dispatchEvent(event, flags) {
2   if (event.dispatch || !event.initialized) {
3     throw new DOMException(INVALID_STATE_ERR) };
4   event.isTrusted = false; event.target = this;
5   return __sDispatch(event, this, flags)
6 }

```

Line-by-Line Closeness. We demonstrate that our JavaScript implementation follows the DOM UI Events standard line-by-line by appealing to the code of the `innerInvoke` function, given below. The `innerInvoke` function is one of the auxiliary functions used by the `Dispatch` algorithm. It is used to execute the listeners for a given event during all three phases of the `Dispatch` algorithm. We illustrate the line-by-line closeness by inlining in comments, for each line of code, its corresponding line in the standard.

```

1 function innerInvoke (event, listeners, phase, legacyOutputDidListenersThrowFlag) {
2   var found = false; // 1. Let found be false.
3   for (var i = 0; i < listeners.length; i++) { // 2. For each listener in
    ↪ listeners...
4     if (listener.removed) continue; // ...whose removed is false:
5     // 2.1. If event's type attribute value is not listener's type, then continue.
6     if (event.type !== listener.type) continue;
7     // 2.2. Set found to true.
8     found = true;
9     // 2.3. If phase is "capturing" and listener's capture is false, then continue.
10    if ((phase === "capturing") && (listener.capture === false)) continue;
11    // 2.4. If phase is "bubbling" and listener's capture is true, then continue.
12    if ((phase === "bubbling") && (listener.capture === true)) continue;
13    // 2.5. If listener's once is true, then remove listener from event's
    ↪ currentTarget attribute value's event listener list.
14    if (listener.once === true) event.currentTarget.removeListener(listener);

```



```

15     ...
16     // 2.10. Call a user object's operation with listener's callback, "handleEvent",
    ↪ event, and event's currentTarget attribute value.
17     execCallback(listener.handleEvent, "handleEvent", event, event.currentTarget);
18     ...
19     // 2.13. If event's stop immediate propagation flag is set, then return found.
20     if (event.stopImmediatePropagation === true) return found;
21 }
22 return found; // 3. Return found
23 }

```

DOM Event Model and the JavaScript Semantics. The interaction between the DOM Dispatch algorithm and the JavaScript semantics may trigger unexpected behaviours if not properly engineered. Consider, for instance, the following function to be used as a handler:

```
function h(ev) { Object.defineProperty(ev, "bubbles", { get: malicious }) }
```

If the programmer registers `h` as an event handler and that event is triggered, the function `malicious` will be implicitly called when the `Dispatch` algorithm tries to resolve the value of the property `bubbles` after the execution of the target phase, because `bubbles` is an accessor property (it does not contain a value, but instead getter/setter functions that are executed on property access/update) and `malicious` is its getter. This behaviour is actually disallowed by the DOM standard, which defines the `bubbles` attribute as read-only, but is exhibited by the DOM engines of Chrome, Edge, Firefox, and Safari. Our reference implementation does not suffer from this problem as we define read-only attributes as non-writable on creation.

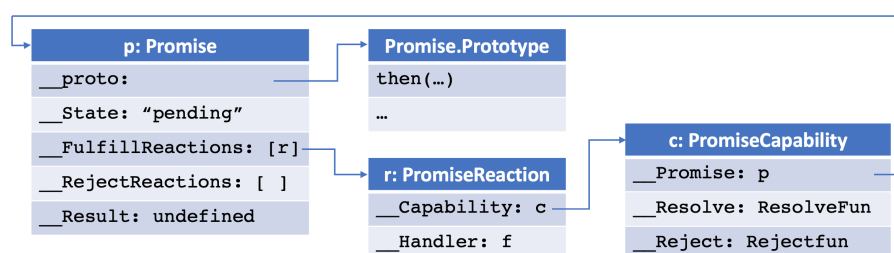
4 JavaScript Promises and `async/await`

Promises were introduced into JavaScript (JS) in the 6th version of the standard [8], in response to the increasing popularity and usefulness of various, often incompatible, custom-made libraries for asynchronous computation. Their addition provided clarity and security to JS developers; in fact, the official Promises API has greatly simplified the creation, combination, and chaining of asynchronous computations, eliminating the so-called *callback hell* of multiple nested callbacks [14], which is extremely difficult to understand and reason about.

A JS Promise, in essence, is the reification of an asynchronous computation that was either already *settled* in the past or still remains to be settled in the future. A promise can be settled successfully, in which case we say that it is *resolved* (the standard also uses the term *fulfilled*), or unsuccessfully, in which case we say that it is *rejected*. If a promise has not been yet settled, we say that it is *pending*.

Promises are often used together with the JS `async/await` API. This API introduces *asynchronous functions*, inside of which one can await on a promise to be fulfilled before proceeding with the current computation. The key point of asynchronous functions is that they *do not block* the execution of their caller function when their execution gets suspended on an `await`; instead, the control is immediately transferred to the caller function, which continues with the execution as if the asynchronous function had simply returned.

This section describes our reference implementations of the JS Promises and `async/await` APIs, as described in sections 25.6, 25.7, and 6.2.3.1 of the 9th version of the ECMAScript standard [9]. Analogously to the DOM reference implementations, these APIs: are implemented directly in JS (ES5 Strict), with the Promises implementation following the standard line-by-line; are thoroughly tested against the latest version of the official ECMAScript test suite [6] (cf. §5); and make use of their dedicated Event Semantics primitives (cf. §2).



■ **Figure 5** Promises Object Graph.

4.1 Promises API

At the core of the Promises API is the promise constructor, `Promise`, which is used for creating new promises. This constructor receives as input an *executor function*, which captures the computation to be performed asynchronously. Executor functions have two arguments: a function `resolve` for stating that the corresponding promise has been resolved, and a function `reject` for stating that it has been rejected. Until one of these functions is called, the corresponding promise is left pending. Consider the following example:

```
function f(v) { console.log(v) };
var p = new Promise( (resolve, reject) => {
  document.getElementById("dv").addEventListener("click", () => { resolve(1) } )
});
p.then(f); console.log(2)
```

This program creates a promise `p`, whose executor function registers the function that resolves the promise as the handler for the `click` event on the DOM element with identifier `dv`. This means that `p` will only get resolved after the user clicks on that DOM element. Afterwards, the program uses the `then` function of the Promises API to register a *fulfill reaction* on the promise `p`, meaning that when/if `p` gets resolved, the function `f` will be scheduled for execution with the argument with which `p` was resolved (in this case, `1`). Reactions are scheduled in a *first-in-first-out* manner every time the current computation terminates or yields control. Hence, the program above will always output the string `21` to the console, regardless of how quickly the user is able to click on the DOM element in question.

Besides the constructor `Promise` and the method `then`, the Promises API provides several other functions for creating, combining, and chaining promises together. The behaviour of these functions/methods is thoroughly described in the ECMAScript standard in pseudo-code. This pseudo-code relies on numerous JavaScript *internal functions*, whose definitions in the ECMAScript standard are also operational, intricate, and intertwined.

The structure of Promise objects is also fairly complex. We illustrate this by giving, in Figure 5, the object graph associated with the promise `p` of the example after the execution of the `then` method, but before the promise gets settled. Each Promise object keeps track of its current state, reactions to be triggered when the promise is resolved/rejected, and its result, in its internal properties `__State`, `__FulfillReactions`, `__RejectReactions`, and `__Result`, respectively. In this case, the promise `p` is in the “pending” state and its result is `undefined`, as it has not been yet resolved. Observe that `f` is registered to execute after `p` using the `then` function in the example; it is not stored directly as a fulfill reaction. Instead, there is a *promise reaction*, `r`, which, in addition to keeping track of `f` in its `__Handler` property, also holds, in its `__Capability` field, a *promise capability* `c`, which keeps track of the promise on whose settlement `f` should be executed (`c.__Promise`), and the `resolve` and `reject` functions given to the executor function of that promise (`c.__Resolve` and `c.__Reject`). In the example, the promise capability `c` contains the promise `p` and the internal resolve and reject algorithms of the standard.

Using the Event Semantics. Our reference implementation of JS promises interacts with the Event Semantics when triggering Promise reactions for a promise that got settled; this is done by the `TriggerPromiseReactions` function. This function is given as input an array of promise reactions and the value with which their corresponding promise was settled (either resolved or rejected). It then iterates over the elements of the array and, for each element, uses the internal function `PromiseReactionJob` to create an anonymous function that will essentially call the handler of the given reaction with the provided value. This anonymous function is then scheduled for execution directly using the wrapper function of the `schedule` primitive of the Event Semantics, as highlighted in line 5 of the following code.

```

1  function TriggerPromiseReactions (reactions, argument) {
2    if (!reactions) return undefined;
3    for (var i = 0; i < reactions.length; i++) {
4      var reactionJob = PromiseReactionJob (reactions[i], argument);
5      __schedule(reactionJob);
6    }
7  }

```

Note that the Event Semantics `schedule` primitive, as defined in §2, adds the given handler to the *end* of the continuation queue. This is consistent with the behaviour of JS Promises described in the standard, Section 8.4.1 [9], which states that pending jobs (essentially, the fulfil and reject reactions) are to be added “at the back of the job queue”.

Line-by-Line Closeness. We demonstrate that our implementation follows the ECMAScript standard line-by-line by appealing to the `FulfillPromise` function, described in the Section 25.4.1.4 of the standard; we give its implementation, annotated with the corresponding lines of the standard. The `FulfillPromise` function is one of the internal functions used by the function `ResolveFun` (shown in Figure 5), which, in turn, is used by promise executors to fulfil their associated promises. The function `FulfillPromise` receives a promise together with the value with which it is to be resolved and proceeds as follows: (1) sets the internal state of the given promise object appropriately; and (2) schedules the promise’s fulfil reactions.

```

1  function FulfillPromise(promise, value) {
2    // 1. Assert: The value of promise's [[State]] internal slot is "pending".
3    Assert(promise.__State === "pending");
4    // 2. Let reactions be the value of promise's [[FulfillReactions]] internal slot.
5    var reactions = promise.__FulfillReactions;
6    // 3. Set the value of promise's [[Result]] internal slot to value.
7    promise.__Result = value;
8    // 4. Set the value of promise's [[FulfillReactions]] internal slot to undefined.
9    promise.__FulfillReactions = undefined;
10   // 5. Set the value of promise's [[RejectReactions]] internal slot to undefined.
11   promise.__RejectReactions = undefined;
12   // 6. Set the value of promise's [[State]] internal slot to "fulfilled".
13   promise.__State = "fulfilled";
14   // 7. Return TriggerPromiseReactions(reactions, value).
15   return TriggerPromiseReactions (reactions, value)
16 }

```

4.2 `async/await`

The `async/await` APIs are defined in Sections 6.3.1 and 25.7 of the 9th version of the ECMAScript standard [9]; they are meant to be used together, as it is only possible to use `await` inside an asynchronous function. Furthermore, the `async/await` APIs directly build on the Promises API in that they make explicit use of JS Promises functions and methods.

In a nutshell, an asynchronous function is a JavaScript function whose execution can *yield*, that is, transfer the control to its calling context without having completed its execution. A call to an asynchronous function is evaluated to a promise that is settled once that function terminates executing: if the function returns, the promise is fulfilled; if the function throws, the promise is rejected. Consider, for instance, the following program:

```
1  async function f () { if (b === true) { return 1 } else { throw 2 } };
2  f().then((v) => { console.log(v) }, (v) => { console.log(v) })
```

(CS1)

Recall that the method `then` receives as input two functions which are registered, respectively, as a fulfil reaction and a reject reaction on the `this` object. Hence, the first function is executed if the promise is fulfilled, whereas the second one is executed if it is rejected. Consequently, in the case of the example, if the global variable `b` is equal to `true`, the program will write 1 to the console, otherwise it will write 2.

As stated above, an asynchronous function can make use of the `await` expression to transfer the control to the calling context. Essentially, the expression `(await e)` evaluates `e` to a promise and suspends the computation of the current function until that promise is settled. Consider, for example, the program below:

```
1  var p = new Promise(function (resolve, reject) { ... });
2  async function g () { return await p };
3  g().then((v) => { console.log(v) }, (v) => { console.log(v) });
```

(CS2)

This time, the asynchronous function `g` awaits on a promise `p`. If/when `p` is settled, `g` returns the value with which it was settled. Suppose, for instance, that `p` is resolved with value 1; in this case, the function `g` returns 1, meaning that its associated promise will also be fulfilled with value 1. Alternatively, suppose that `p` is rejected with value 1; then, `g` throws 1, meaning that its associated promise will also be rejected with value 1. In both cases, the program will simply write 1 to the console.

Line-by-line Closeness. For `async/await`, we depart from our line-by-line closeness approach. The reason is that this would require JSIL to support first-order execution contexts, which, in turn, would constitute a considerable engineering effort, including changing the internal representation of execution contexts and extending JSIL with various primitives for their manipulation. Instead, we opted for a more lightweight, compilation-based, approach that still correctly models the `async/await` behaviour described in the standard.

Compiling `async/await` to ES5 Strict. As `async` and `await` fundamentally change the control flow behaviour of the language, they cannot be simply implemented as libraries. Hence, we introduce a pre-compilation step that translates these constructs to ES5 Strict. Expectedly, the compiled programs use the Promise constructor to create the promise associated with the execution of the asynchronous function being compiled. The key case of the compiler, given below, corresponds to the default translation¹ of asynchronous functions:

$$\mathcal{C}\langle \text{async function}(\bar{x})\{s\} \rangle \triangleq \text{function}(\bar{x}) \{$$

$$\quad \text{return new Promise(function(resolve, reject) \{$$

$$\quad \quad \text{try } \{ \mathcal{C}_a\langle s \rangle; \text{ resolve(undefined) } \} \text{ catch}(e) \{ \text{reject}(e) \}$$

$$\quad \quad \}$$

$$\quad \}$$

¹ If an asynchronous function can return from a `finally` block, the settling of its associated promise must be deferred to that `finally` block, making the compilation of `return` statements more complex.

Essentially, an asynchronous function is compiled to a normal ES5 Strict function that simply creates a promise `p` and returns it. The body of the original function is run inside the executor of the promise. Additionally, we make use of an auxiliary compiler \mathcal{C}_a to rewrite `return` statements inside the body of the original function so that they are replaced by a call to `resolve` followed by an empty `return`. Hence, the function `f`, given in Code Snippet 1, is compiled to:

```

1  function f () {
2    return new Promise (function (resolve, reject) {
3      try { if (b === true) { resolve(1); return } else { throw 2 } }
4        catch (x) { reject(x); return }
5    })
6  }

```

The compilation of the `await p` expression is more involved. Concretely, the compiled code calls the wrapper function for the Event Semantics `await` primitive with the argument `getPredicate(p)`, which corresponds to a function that evaluates to `true` once the promise `p` has been settled. Given the definition of `await` in §2, this precisely corresponds to the core behaviour of the JS `await`. Then, the compiled code checks if the promise was fulfilled: if so, it continues with the execution normally; if not, it throws the value with which the promise was rejected. Below, we illustrate the compilation of the function `g`, given in Code Snippet 2.

```

1  function g () {
2    return new Promise (function (resolve, reject) {
3      try {
4        __await(getPredicate(p));
5        if(p.__State === "resolved") { resolve(p.__Result) } else { throw p.__Result }
6      } catch (x) { reject(x); return }
7    })
8  }

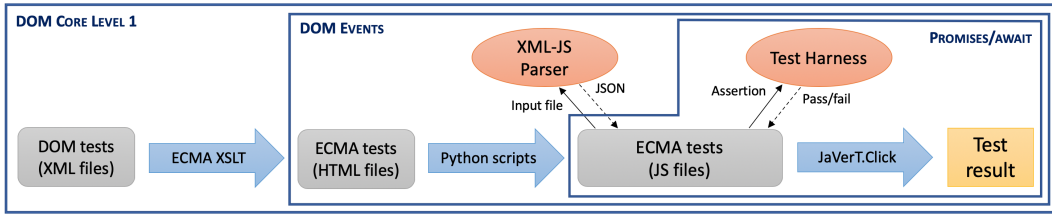
```

5 Evaluation

We show that our reference implementations of DOM Core Level 1, DOM UI Events, JS Promises, and `async/await` are trustworthy by passing all applicable tests from their official test suites [46, 50, 6] in JaVerT.Click. In doing so, we discover coverage gaps in the DOM Core Level 1 and UI Events test suites and create additional tests to complete their coverage. We demonstrate that JaVerT.Click can reason about real-world JS code by creating a comprehensive symbolic test suite for the `events` module of the `cash` library [55], a widely-used alternative for jQuery. Our symbolic testing establishes bounded correctness of several essential properties of the library and reveals two subtle, previously unknown bugs. We also symbolically test the `p-map` library [35], which adds an extra layer of functionality on top of JS promises. We achieve 100% line coverage and discover one bug. All three bugs discovered by JaVerT.Click in `cash` and `p-map` have been reported and have since been fixed.

5.1 Testing the Reference Implementations

To run the test suites, we establish a common testing infrastructure, illustrated below. The tests for Promises and `async/await` are written in JS. To run them in JaVerT.Click, we only need to compile the ECMAScript test harness together with the tests. The tests for DOM Events, in contrast, are written in HTML and contain JS scripts enclosed by the `<script>` tag. Using Python scripts, we first isolate this code into a JS test file, then add to it the JSON object obtained from the appropriate input XML file using the `xml-js` parser [58]. Finally, as the DOM Core Level 1 tests are written in XML, we additionally have to first transform them into HTML tests using XSLT.



We present the results of testing our reference implementations against their appropriate official test suites. For each implementation, we provide the number of: available tests in the test suite; applicable tests; and passing tests. Additionally, for three of the test suites, we give: its computed line coverage; the number of untested lines in its standard; and the number of additional tests that we created to complete its coverage. Given that we pass all of the applicable tests, which have substantial coverage of their respective standards, for all four APIs, we have strong confidence that our reference implementations are indeed correct. Interestingly, the test suite repository for DOM Events also provides the testing results for four browsers: Chrome, Edge, Firefox and Safari [57]. These results show that no single browser fully passes the test suite, and that, out of the 56 tests that we pass, 12 fail in at least one of the four browsers.

	Core Level 1	Events	Promises	async/await
Available Tests	527	83	474	86
Applicable Tests	527	56	344	68
Passing Tests	527	56	344	68
Test Suite Line Coverage	98.14%	97.45%	98.76%	N/A
Number of Untested Lines	13	8	5	N/A
Additional Tests	5	3	N/A	N/A

For three of the test suites, some tests need to be filtered out due to the coverage of either our reference implementations or JaVerT.Click. For DOM Events, we filter out 1 test that uses ES6 classes, 5 that use the `postMessage` API, 2 that use the `AJAX` API, and 19 that use unsupported CSS features (scrolling and animation). For Promises, most filtered tests (106/130) are due to ES6 Symbol iteration; once we support this feature, these tests should pass as similar tests that use Array iteration already pass. We also filter out 21 tests that use other unsupported ES6 features (classes, reflection, and proxies), and 3 that require non-strict mode. For `async/await`, we filter out 14 tests that use ES6 default arguments and 4 that use ES6 generators.

When it comes to test suite coverage, we observe that it is comprehensive, but incomplete. We have inspected the filtered tests for DOM Events and JS Promises and believe that they would not trigger the missing lines. Note that we are not able to perform a proper coverage analysis for `async/await`, as we do not follow its description in the standard line-by-line.

Observations. We have found the ECMAScript standard to be written and tested with a higher degree of rigour than the DOM Living Standard. It is self-contained and precise, with no implicit assumptions and discrepancies between the standard and the test suite.

The DOM Living Standard, in contrast, uses features from other standards, such as HTML and the Shadow DOM [26], without providing any intuition. This meant that we needed to understand multiple standards written in different formats and had to read

substantial additional documentation (e.g., the Mozilla Web Docs [25]) in order to model the API behaviour correctly. Additionally, the DOM Living Standard interfaces do not have a well-defined scope. For instance, the standard makes clear that every `EventTarget` object has an associated event listener list, but this list is not declared as an attribute of the `EventTarget` interface. This can lead to different interpretations by implementors. Finally, we found a few discrepancies between the DOM Standard and the official test suites that are likely to cause difficulties for implementors: for example:

- In DOM Core Level 1, on setting `Attr.value`, the standard only states that a `Text` node with the unparsed contents of the provided value should be created; the tests additionally require that this text node be inserted as a child node of the attribute.
- In DOM Events, for `Event.isTrusted`, the standard defines the `isTrusted` property of the `Event` interface to be a boolean that is used to indicate whether or not the `dispatchEvent` function was used; the tests specifically require the `isTrusted` property to be an accessor property and to have a dedicated getter.

5.2 Symbolic Testing of the `cash` Library

The `cash` library [55] is a jQuery alternative for modern browsers that provides jQuery-style syntax for manipulating the DOM. Its main goal is to remain as small as possible, while still staying (mostly) compatible with jQuery and providing its users with a similar set of features. Moreover, it exhibits better performance than jQuery, as it dominantly relies on native browser events rather than on a custom event model. It has a growing community of users, with more than 10K weekly downloads and 735K overall downloads on npm [56], and more than 4.4K stars on GitHub [55].

We focus our analysis on the `events` module of `cash`, which provides a mechanism for creating and manipulating DOM events, offering additional functionalities and greater level of control with respect to the native DOM event model. This module has five main and twelve auxiliary functions. Here, we focus on the main functions, presented below:

- .on:** `ele.on(e,h)` registers the handler `h` for an event `e` on the element `ele`;
- .off:** `ele.off(e,h)` deregisters the handler `h` for the event `e` on the element `ele`;
- .one:** `ele.one(e,h)` behaves the same as `.on`, except that `h` can be triggered only once and is automatically deregistered afterwards;
- .ready:** `ele.ready(f)` executes the function `f` after ensuring that the entire document content has been loaded successfully;
- .trigger:** `ele.trigger(e)` triggers the handlers for an event `e` on the element `ele`.

The `cash` library comes with a concrete test suite, which has 95.52% overall line coverage. The 18 tests for the `events` module contain 288 lines of code. Their coverage of `.on` is 76.92%, of `.trigger` is 93.75%, of `.ready` is 0% and of the main auxiliary function of `.on` is 81.82%; the remaining functions have 100% coverage. We complete the coverage of the concrete test suite for the `events` module by writing five additional concrete tests.

5.2.1 Bounded Correctness

We create a symbolic test suite for the `events` module of `cash`, with two goals in mind: (1) achieving 100% line coverage for all event-related functions; and (2) establishing bounded correctness of several essential properties. We achieve both goals using just eight symbolic tests. In Table 1, we give, for these tests, their execution time (Time, in seconds) and the number of executed JSIL commands (JSIL Cmds). Each test, additionally, has an overhead of 4.454 seconds, 9 lines of code, and 899,390 executed commands due to the setup of the

initial heap and auxiliary testing functions. We single out four tests, which capture important properties that the `events` module should respect; the remaining ones are grouped together as `other`, as they offer little additional insight. These four tests are:

rHand : If a handler has been executed, then it must have previously been registered.

sHand : If a single handler has been registered to a given event using `.on`, then that is the *only* handler that can be executed for that event. This test has revealed two bugs in the `events` module of `cash`, discussed in detail in §5.2.2.

tOne : If a single handler has been registered to a given event using `.one`, then that handler can be executed for that event *only once*.

tOff : If a handler registered to an event is deregistered using `.off`, then that handler can no longer be executed for that event.

The tests establish that these properties hold *for all* events (strings) up to length 20. The bound 20 has been chosen as the length of the longest property of the JavaScript initial heap, `propertyIsEnumerable`. It can be adjusted in the tests themselves: the running times will be bound-linear for `rHand`, `tOne`, and `tOff`, which use one symbolic event; and bound-quadratic for `sHand`, which uses two.

The obtained results demonstrate that symbolic testing is far superior to concrete testing: our symbolic test suite has greater coverage, 29% fewer lines of code, and, most importantly, provides much stronger correctness guarantees that are beyond the limit of concrete testing.

5.2.2 The Discovered Bugs

As part of its effort to remain minimal, the `cash` library, unlike jQuery, does not implement its own event model. Instead, it heavily relies on the event model of the browser. However, the semantics of events in `cash` differs from that of the browser events. For example, `cash` enforces that all user-defined focus-related handlers bubble, by *redirecting* handler registration (via `.on` or `.one`) and deregistration (via `.off`) for the `'focus'/'blur'` events to `'focusin'/'focusout'` instead. The redirection is implemented as follows: any event that is passed to the `.on`, `.one`, and `.off` functions is first processed by the `getEventTypeBubbling` function:

```
function getEventTypeBubbling(e) { return eventsFocus[e] || e }
```

which is intended to substitute `'focus'` by `'focusin'` and `'blur'` by `'focusout'`, while keeping other events intact, by indexing the `eventsFocus` object

```
var eventsFocus = { focus: 'focusin', blur: 'focusout' }.
```

with the event `e`. This indexing is meant to return a string, which is then processed using `String.prototype.split`. This implementation, however, causes two subtle bugs, discovered by the `sHand` test, whose stylised code, with detailed inlined explanations, is given below:

```
1 var count = 0, ele = $(' .event '); // Initialise counter and target element
2
3 function h () { count++ } // Handler counts the number of times it was called
```

■ **Table 1** Symbolic Test Suite for the `events` module of `cash`.

Test Name	rHand	sHand	tOne	tOff	other	Total
Time (s)	5.54	144.38	24.35	22.87	42.20	239.34
JSIL Cmds	1,468,907	38,240,506	9,288,337	9,400,471	14,150,893	72,549,114


```

4
5 // Create two symbolic events, e1 and e2, of maximum length 20
6 var e1 = symbStr(20), e2 = symbStr(20);
7
8 // Register the handler for e1 on ele, then trigger e2 on ele
9 ele.on(e1, h); ele.trigger(e2);
10
11 Assert(
12   // Handler was executed only once, if e1 and e2 were equal and non-empty,
13   (count === 1 && e1 === e2 && e1 !== "") ||
14   // and was not executed otherwise.
15   (count === 0 && (e1 !== e2 || e1 === ""))
16 );

```

Bug 1: Overlooked Prototype Inheritance. The first set of counter-examples demonstrates that `cash` throws a native JavaScript type error when executing `ele.on(e1, h)` if

$$e1 \in \{\text{'constructor'}, \text{'hasOwnProperty'}, \text{'isPrototypeOf'}, \text{'propertyIsEnumerable'}, \text{'toLocaleString'}, \text{'toString'}, \text{'valueOf'}\}.$$

Recall that the function `getEventTypeBubbling` indexes the `eventsFocus` object to redirect focus-related events. Indexing objects as key-value maps, however, may return unexpected values, as shown in [32]: e.g., `eventsFocus['valueOf']` returns the function object found at `Object.prototype.valueOf`, as the `'valueOf'` property is not in the `eventsFocus` object itself, but is in its prototype. Then, since that function object has no `split` property in its prototype chain, the subsequent call to `.split` throws a native JavaScript type error.

Bug 2: Unintended Event Triggering. The second set of counter-examples demonstrates that the final correctness assertion of the `sHand` test does not hold if

$$(e1, e2) \in \{(\text{'blur'}, \text{'blur'}), (\text{'focus'}, \text{'focus'}), (\text{'blur'}, \text{'focusout'}), (\text{'focus'}, \text{'focusin'})\}.$$

In particular, for the first two counter-examples, the handler is not executed even though `e1` and `e2` are equal, whereas, for the second two, it is executed despite `e1` and `e2` being different. This bug is also caused by the redirection done in the `getEventTypeBubbling` function. Precisely, this redirection is applied in the `.on`, `.one`, and `.off` functions, but not in the `.trigger` function, effectively meaning that user-registered handlers for `'focus'` and `'blur'` can respectively be triggered *only* via `'focusin'` and `'focusout'` instead. This is admittedly not intended, and it results from the simplification of the corresponding jQuery mechanisms.

Both bugs have been reported to the developers of `cash`,² and have since been fixed. The first bug also exists in jQuery, where it will be corrected for the upcoming 4.0 version.³

5.3 Symbolic Testing of the `p-map` Library

The `p-map` library [35] is a small JavaScript library that extends the functionality of JavaScript promises with the ability to concurrently map over pending promises. It has more than 10M weekly downloads and 825M overall downloads on npm [36], and 532 stars on

² <https://github.com/kenwheeler/cash/issues/317>, 318

³ <https://github.com/jquery/jquery/issues/3256>

GitHub [35]. It calls both the JavaScript Promises and JavaScript `async/await` APIs. We performed symbolic testing of `p-map`, where we achieved 100% line coverage and discovered a bug that allowed the number of concurrently handled promises to go above its declared maximum due to the library using non-integer numbers. This bug has been reported to and fixed by the developers of `p-map`.⁴ For space reasons, we delay the full account of our analysis of `p-map` to a future publication.

6 Related Work

We believe we are the first to provide a general infrastructure for symbolic analysis of modern event-driven Web applications. There has been prior work on formalising and analysing specific event-driven Web APIs, such as DOM UI Events [29, 19] and JavaScript Promises [22, 1], as well as Node.js events [21, 23]. However, to the best of our knowledge, there is no prior work on formalising the JavaScript `async/await` API. Hence, we focus the discussion on: **(1)** axiomatic and operational semantics for DOM Core Level 1; **(2)** operational semantics for DOM Events; **(3)** operational semantics for JavaScript Promises; and **(4)** symbolic execution for JavaScript programs that interact with the DOM.

Axiomatic/Operational Semantics of DOM Core Level 1. Based on context logic [5], Smith et al. introduced an axiomatic semantics [15] for a small fragment of DOM Core Level 1, proving it sound with respect to their operational semantics. In his PhD thesis [34], Smith extended this axiomatic semantics to all fundamental interfaces of DOM Core Level 1, including live collections and fine-grained reasoning about various types of DOM nodes, omitting only a minor part on the extended interfaces. This axiomatic semantics follows the DOM standard closely, but has not been implemented, and there has been no work on using this semantics to reason about real-world JavaScript programs that interact with the DOM.

Several operational semantics for different fragments and adaptations of DOM Core Level 1 were proposed for various types of analysis, such as information flow control [24, 30], type systems [42] and abstract interpretation [18], targeting JS programs that interact with the DOM. These papers, however, do not aim to establish a trusted formal representation of the DOM using which others can build their own program analyses; instead, they provide a DOM representation specific to their kind of analyses. In contrast, our DOM Core Level 1 JS reference implementation has been designed to be trusted in that it follows the text of the standard line-by-line and passes all 525 tests of the official test suite [46]. This, combined with its extensive use in the symbolic testing of the `cash` library, gives us confidence that others will be able to use it for their analysis of JS programs calling the DOM.

Operational Semantics for DOM Events. In this context, the work closest to ours is [19], which presents the first operational model for reasoning about DOM events. This model consists of a Scheme [38] reference implementation of DOM UI events and is used to prove meta-properties of the DOM semantics, such as the immutability of the propagation path during the execution of the `Dispatch` algorithm. The authors justify their reference implementation by annotating the paragraphs of the standard with links to the relevant definitions and reduction rules in their implementation, and by comparing its behaviour with various browser implementations using randomly generated test cases. The implementation, however, is not tested against the official DOM Events test suite and does not have a line-by-line correspondence with the text of the standard.

⁴ <https://github.com/sindresorhus/p-map/issues/26>

There are multiple tools for analysing event-driven JavaScript programs based on different types of program analyses, such as information flow control [29, 45], type systems [28], and abstract interpretation [27]. Of these tools, only [29] comes with a formal semantics of DOM events. Concretely, the authors propose a simplified DOM event semantics instrumented with a sound information-flow monitor, and implement the monitor instrumentation on top of Webkit [41], the browser engine used by Safari. The proposed semantics is, however, only intended for illustrative purposes as it does not include a number of event-related features, such as interaction with shadow trees, slotables, and touch/related targets. In contrast, our reference implementation of DOM Events does not simplify the standard and passes 56 tests of the official test suite (100% of the appropriate tests, given our current coverage).

A Core Semantics for JavaScript Promises. Madsen et al. [22] were the first to propose a formal core calculus for reasoning about JavaScript (JS) promises. Concretely, they introduce λ_p , an extension of the small core JavaScript calculus, λ_{JS} [17], with dedicated syntactic constructs for promise creation and manipulation. The authors give the formal semantics of λ_p and show how it can be used to encode promise operations not directly supported in the syntax (e.g. `catch` and `then`). The paper further introduces the concept of *promise graphs*, a program artifact used by the authors to explain promise-related errors. Later, Alimadadi et al. [1] extend promise graphs to take into account previously unmodelled ES6 features, such as default reactions, exceptions, `race` and `all`. Using the extended promise graphs, the authors develop PROMISEKEEPER, a dynamic analysis tool built on top of JALANGI [33] for finding and explaining promise-related bugs in JS code.

The λ_{JS} -calculus [17] was justified by a desugaring function from ES5 that has been tested against the official Test262 test suite [6]. In contrast, λ_p does not come with a desugaring function from ES6 to λ_p and hence has not been tested against the promises-related part of Test262. Whilst λ_p is mainly used to explain buggy behaviours related to the misuse of JS promises, our goal was to create a trusted reference implementation of JS promises that models their semantics precisely in order to enable various types of analysis for JS programs that use promises, including the symbolic testing presented in the paper. For this reason, we took great care in justifying its correctness.

Symbolic Execution for the DOM. Symbolic reasoning about the DOM in the literature is mostly focussed on bug-finding and/or automatic concrete test generation. For example, CONFIX [10] uses concolic execution to generate DOM fixtures that allow high-coverage testing of JavaScript functions that use the DOM; however, it does not support DOM events and dynamically generated code using `eval` that interacts with the DOM. V-DOM [59] creates test suites by analysing both server- and client-side code, but only considers handlers that were registered statically (via HTML code, e.g. `<button onclick="myFunction()"/>`) and does not support dynamic handler registration (via `addEventListener()`).

There are several tools focussed on finding dependencies between event handlers, such as SYMJS [20] and JSDEP [37], which then use this information to automatically generate tests in the form of event triggering sequences. SYMJS identifies handler dependencies by performing a dynamic write-read analysis. However, its representation of the DOM is not entirely consistent with the standard: e.g., text inputs and radio boxes are represented symbolically either as strings or numbers, rather than objects. JSDEP implements the first constraint-based declarative program analysis for computing dependencies between event handlers. This approach is shown to be effective, but no soundness guarantees are provided.

While the goals of these tools are different from ours, there is room for comparison. In particular, some of them do not follow the DOM standard (e.g., SYMJS relies on HTMLUnit [16], which provides its own implementation of the DOM event dispatch algorithm) and none offer a justification with respect to their representation of the DOM. In contrast, we provide complete, trustworthy reference implementations of DOM Core Level 1 and UI Events that follow the standard line-by-line and pass all of the applicable official tests. Importantly, these tools do not appear to be able to reason about events whose *type* is symbolic. We believe that this is one of the advantages of our work, as it allows us to write few symbolic tests to achieve broad coverage. It also enables us to provide bounded correctness guarantees of library properties, which, to our knowledge, has not been done before, and which is certainly beyond the reach of either manually- or automatically-generated concrete test suites. On the other hand, the above-mentioned tools do generate their concrete test suites automatically, while in JaVerT.Click, the developers have to write symbolic tests themselves.

7 Conclusions and Future Work

We have introduced a Core Event Semantics that is simple in design, yet expressive enough to capture the essence of three fundamental, complex event-related APIs, namely DOM UI Events, JavaScript Promises and `async/await`. To accompany the Core Event Semantics, we have created reference implementations of these three APIs, as well as a reference implementation of DOM Core Level 1, which underpins DOM UI Events. Our reference implementations are trusted, in the sense that all except that of `async/await` follow their respective standards line-by-line, and all are thoroughly tested against their official test suites. Together, the Core Event Semantics and the reference implementations form a trusted infrastructure that enables symbolic analysis of modern event-driven Web programs.

We have demonstrated that our infrastructure can be used in practice by implementing the Core Event Semantics, closely following the theory, on top of JaVerT 2.0, a state-of-the-art tool for JavaScript symbolic analysis. We have used the resulting tool, JaVerT.Click, to symbolically test two real-world libraries: `cash` and `p-map`, both with with 100% line coverage, establishing bounded correctness of several important properties and discovering three bugs in the process. To our knowledge, this is the first time that reasoning about multiple event-based APIs is supported either in a single formalism or in a single tool.

As part of the overall testing process, we have additionally discovered coverage gaps in the official test suites of DOM Core Level 1 and DOM UI Events, as well as in the concrete test suite of `cash`, and have created appropriate concrete tests that address these gaps.

We plan to extend this work in several directions. First of all, following the methodology that we have introduced in this paper, we will add support for other event-based APIs, such as the File [51], `postMessage` [54], and the Web Workers APIs [52], to the Core Event Semantics and JaVerT.Click. For each new API, this amounts to providing a trusted reference implementation in JavaScript, and extending the Event Semantics with any new appropriate event primitives that may be required. For instance, supporting the Web Workers API will require the Event Semantics to be extended with basic message-passing facilities. Our over-arching goal is to create a minimal event model expressive enough to reason about all widely-used Web APIs natively supported by major browsers.

We also intend to analyse further real-world libraries that are clients of our supported APIs. For example, PreactJS [39], a fast and light alternative to ReactJS [40], appears to be an excellent first target, as it is relatively small yet very successful, and is already being used by several major industrial players.

Another avenue to explore, given our trusted infrastructure, would be how to extend the full verification facilities of JaVerT.Click in order to be able to prove both meta-properties of the APIs themselves as well as correctness properties of programs that interact with the DOM and/or use event-related APIs.

We will also implement the Event Semantics as a layer on top of Gillian [11], our new multi-language platform for compositional symbolic analysis, by instantiating the Event Semantics with Gillian’s intermediate language, GIL. There, in addition to JS code, we plan to reason about WebAssembly and Rust code that interacts with various event-based APIs.

Finally, we plan to design a policy language that would allow the developers to specify the event sequences of interest, given the programs they would like to analyse. These policies might play a role in automatically generating tests, as in the discussed related work [20, 37], but also in the broader context of symbolic analysis, where they would limit the branching that arises from exploring all possible event sequences triggered by the environment.

References

- 1 S. Alimadadi, D. Zhong, M. Madsen, and F. Tip. Finding Broken Promises in Asynchronous JavaScript Programs. *PACMPL*, 2(OOPSLA):162:1–162:26, 2018.
- 2 R. Baldoni, E. Coppa, D. Cono D’Elia, C. Demetrescu, and I. Finocchi. A Survey of Symbolic Execution Techniques. *ACM Computing Surveys*, 51(3):50:1–50:39, 2018.
- 3 C. Cadar, P. Godefroid, S. Khurshid, C. S. Păsăreanu, K. Sen, N. Tillmann, and W. Visser. Symbolic Execution for Software Testing in Practice: Preliminary Assessment. In *ICSE*, 2011.
- 4 C. Cadar and K. Sen. Symbolic Execution for Software Testing: Three Decades Later. *Commun. ACM*, 56:82–90, 2013.
- 5 C. Calcagno, P. Gardner, and U. Zarfaty. Context logic and tree update. In *POPL*, 2005.
- 6 ECMA TC39. Test262 Test Suite. <https://github.com/tc39/test262>, visited 05/2020.
- 7 ECMA TC39. The ECMAScript Standard. <https://www.ecma-international.org/publications/standards/Ecma-262.htm>, visited 05/2020.
- 8 ECMA TC39. The ECMAScript Standard - 6th Edition. <http://www.ecma-international.org/ecma-262/6.0/>, visited 05/2020.
- 9 ECMA TC39. The ECMAScript Standard - 9th Edition. <http://www.ecma-international.org/ecma-262/9.0/>, visited 05/2020.
- 10 A. M. Fard, A. Mesbah, and E. Wohlstadtter. Generating Fixtures for JavaScript Unit Testing (T). In *ASE*, 2015.
- 11 J. Fragoso Santos, P. Maksimović, S.-E. Ayoun, and Philippa G. Gillian, Part 1: A Multi-language Framework for Symbolic Execution. In *PLDI*, 2020.
- 12 J. Fragoso Santos, P. Maksimović, T. Grohens, J. Dolby, and P. Gardner. Symbolic Execution for JavaScript. In *PPDP*, 2018.
- 13 J. Fragoso Santos, P. Maksimović, G. Sampaio, and P. Gardner. JaVerT 2.0: Compositional Symbolic Execution for JavaScript. *PACMPL*, 3(POPL):66, 2019.
- 14 K. Gallaba, A. Mesbah, and I. Beschastnikh. Don’t Call Us, We’ll Call You: Characterizing Callbacks in Javascript. In *ESEM*, 2015.
- 15 P. Gardner, G. Smith, M. J. Wheelhouse, and U. Zarfaty. Local Hoare Reasoning about DOM. In *PODS*, 2008.
- 16 Gargoyl Software Inc. HTMLUnit. <http://htmlunit.sourceforge.io/>, visited 05/2020.
- 17 A. Guha, C. Saftoiu, and S. Krishnamurthi. The Essence of Javascript. In *ECOOP*, 2010.
- 18 S. Holm Jensen, M. Madsen, and A. Møller. Modeling the HTML DOM and Browser API in Static Analysis of JavaScript Web Applications. In *ESEC/FSE*, 2011.
- 19 B. S. Lerner, M. J. Carroll, D. P. Kimmel, H. Quay-De La Vallee, and S. Krishnamurthi. Modeling and Reasoning about DOM Events. In *WebApps*, 2012.
- 20 G. Li, E. Andreasen, and I. Ghosh. SymJS: Automatic Symbolic Testing of JavaScript Web Applications. In *FSE*, 2014.

- 21 M. Loring, M. Marron, and D. Leijen. Semantics of asynchronous javascript. In *DLS*, 2017.
- 22 M. Madsen, O. Lhoták, and F. Tip. A Model for Reasoning about JavaScript Promises. *PACMPL*, 1(OOPSLA):86:1–86:24, 2017.
- 23 M. Madsen, F. Tip, and O. Lhotak. Static Analysis of Event-Driven Node.js JavaScript Applications. In *OOPSLA*, 2015.
- 24 A. Almeida Matos, J. Fragoso Santos, and T. Rezk. An Information Flow Monitor for a Core of DOM - Introducing References and Live Primitives. In *TGC*, 2014.
- 25 Mozilla. MDN Web Docs. <http://developer.mozilla.org/en-US/>, visited 05/2020.
- 26 Mozilla. Using Shadow DOM. https://developer.mozilla.org/en-US/docs/Web/Web_Components/Using_shadow_DOM, visited 05/2020.
- 27 C. Park, S. Won, J. Jin, and S. Ryu. Static Analysis of JavaScript Web Applications in the Wild via Practical DOM Modeling (T). In *ASE*, 2015.
- 28 J. Gibbs Politz, S. A. Eliopoulos, A. Guha, and S. Krishnamurthi. ADSafety: Type-Based Verification of JavaScript Sandboxing. In *USENIX Security Symposium*, 2011.
- 29 V. Rajani, A. Bichhawat, D. Garg, and C. Hammer. Information Flow Control for Event Handling and the DOM in Web Browsers. In *CSF*, 2015.
- 30 Alejandro Russo, Andrei Sabelfeld, and Andrey Chudnov. Tracking Information Flow in Dynamic Tree Structures. In *ESORICS*, 2009.
- 31 G. Sampaio, J. Fragoso Santos, P. Maksimović, and P. Gardner. A Trusted Infrastructure for Symbolic Analysis of Event-Driven Web Applications (Technical Report). <https://vtss.doc.ic.ac.uk/publications/Sampaio2020Trusted.pdf>, visited 05/2020.
- 32 J. Fragoso Santos, P. Maksimović, D. Naudžiūnienė, T. Wood, and P. Gardner. JaVerT: JavaScript Verification Toolchain. *PACMPL*, 2(POPL):50:1–50:33, 2018.
- 33 K. Sen, S. Kalasapur, T. G. Brutch, and S. Gibbs. Jalangi: A Selective Record-Replay and Dynamic Analysis Framework for JavaScript. In *ESEC/FSE'13*, 2013.
- 34 G. Smith. *Local reasoning about Web programs*. PhD thesis, Imperial College, UK, 2011.
- 35 S. Sorhus. p-map (GitHub). <https://github.com/sindresorhus/p-map>, visited 05/2020.
- 36 S. Sorhus. p-map (npm). <https://www.npmjs.com/package/p-map>, visited 05/2020.
- 37 C. Sung, M. Kusano, N. Sinha, and C. Wang. Static DOM Event Dependency Analysis for Testing Web Applications. In *FSE*, 2016.
- 38 Scheme Team. The Revised Report on the Algorithmic Language Scheme. <https://schemers.org/Documents/Standards/R5RS/r5rs.pdf>, visited 05/2020.
- 39 The PreactJS Team. PreactJS library. <http://preactjs.com>, visited 05/2020.
- 40 The ReactJS Team. ReactJS library. <http://reactjs.org>, visited 05/2020.
- 41 The WebKit Team. WebKit Browser Engine. <https://webkit.org>, visited 05/2020.
- 42 Peter Thiemann. A Type Safe DOM API. In *DBPL*, 2005.
- 43 E. Torlak and R. Bodík. Growing Solver-Aided Languages with Rosette. In *Onward!*, 2013.
- 44 E. Torlak and R. Bodík. A Lightweight Symbolic Virtual Machine for Solver-Aided Host Languages. In *PLDI*, 2014.
- 45 M. Vanhoef, W. De Groef, D. Devriese, F. Piessens, and T. Rezk. Stateful Declassification Policies for Event-Driven Programs. In *CSF*, 2014.
- 46 W3C. DOM Core Level 1 Official Test Suite. <http://www.w3.org/2004/04/ecmascript/level1/core/>, visited 05/2020.
- 47 W3C. DOM Core Level 1 Specification. <http://www.w3.org/TR/1998/REC-DOM-Level-1-19981001/level-one-core.html>, visited 05/2020.
- 48 W3C. DOM Core Level 2 Specification. <http://www.w3.org/TR/2000/REC-DOM-Level-2-Core-20001113/>, visited 05/2020.
- 49 W3C. DOM Core Level 3 Specification. <http://www.w3.org/TR/2004/REC-DOM-Level-3-Core-20040407/>, visited 05/2020.
- 50 W3C. DOM Events Official Test Suite. <https://github.com/web-platform-tests/wpt/tree/master/dom/events>, visited 05/2020.
- 51 W3C. File API. <http://www.w3.org/TR/FileAPI/>, visited 05/2020.

- 52 W3C. HTML Standard. <http://html.spec.whatwg.org/multipage/workers.html#workers>, visited 05/2020.
- 53 WHATWG. DOM API Specification. <http://dom.spec.whatwg.org>, visited 05/2020.
- 54 WHATWG. The postMessage API. <https://html.spec.whatwg.org/multipage/web-messaging.html#posting-messages>, visited 05/2020.
- 55 K. Wheeler, S. Shaw, and F. Spampinato. cash (GitHub). <https://github.com/kenwheeler/cash>, visited 05/2020.
- 56 K. Wheeler, S. Shaw, and F. Spampinato. cash (npm). <https://www.npmjs.com/package/cash-dom>, visited 05/2020.
- 57 wpt.fyi. Events Browser Compliance. <http://wpt.fyi/results/dom/events>, visited 05/2020.
- 58 Y. Nashwaan. xml-js: Converter Utility between XML Text and Javascript Objects/JSON Text. <http://www.npmjs.com/package/xml-js>, visited 05/2020.
- 59 Y. Zou, Z. Chen, Y. Zheng, X. Zhang, and Z. Gao. Virtual DOM Coverage: Drive an Effective Testing for Dynamic Web Applications. *ISSTA*, 2014.

The Duality of Subtyping

Bruno C. d. S. Oliveira

The University of Hong Kong, China
bruno@cs.hku.hk

Cui Shaobo

University of California San Diego, CA, USA
cuishaobo@gmail.com

Baber Rehman

The University of Hong Kong, China
brehman@cs.hku.hk

Abstract

Subtyping is a concept frequently encountered in many programming languages and calculi. Various forms of subtyping exist for different type system features, including intersection types, union types or bounded quantification. Normally these features are designed independently of each other, without exploiting obvious similarities (or dualities) between features.

This paper proposes a novel methodology for designing subtyping relations that exploits duality between features. At the core of our methodology is a generalization of subtyping relations, which we call *Duotyping*. *Duotyping* is parameterized by the mode of the relation. One of these modes is the usual subtyping, while another mode is supertyping (the dual of subtyping). Using the mode it is possible to generalize the usual rules of subtyping to account not only for the intended behaviour of one particular language construct, but also of its dual. *Duotyping* brings multiple benefits, including: shorter specifications and implementations, dual features that come essentially for free, as well as new proof techniques for various properties of subtyping. To evaluate a design based on *Duotyping* against traditional designs, we formalized various calculi with common OOP features (including union types, intersection types and bounded quantification) in Coq in both styles. Our results show that the metatheory when using *Duotyping* does not come at a significant cost: the metatheory with *Duotyping* has similar complexity and size compared to the metatheory for traditional designs. However, we discover new features as duals to well-known features. Furthermore, we also show that *Duotyping* can significantly simplify transitivity proofs for many of the calculi studied by us.

2012 ACM Subject Classification Software and its engineering → Object oriented languages

Keywords and phrases DuoTyping, OOP, Duality, Subtyping, Supertyping

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2020.29

Supplementary Material ECOOP 2020 Artifact Evaluation approved artifact available at
<https://doi.org/10.4230/DARTS.6.2.8>.
<https://github.com/baberrehman/coq-duotyping>

Funding Funded by Hong Kong Research Grant Council projects number 17210617 and 17209519.

Acknowledgements We thank the anonymous reviewers for their helpful comments.

1 Introduction

Subtyping is a concept frequently encountered in many programming languages and calculi. It is also a pervasive and fundamental feature in Object-Oriented Programming (OOP). Various forms of subtyping exist for different type system features, including *intersection types* [6], *union types* [6] or *bounded quantification* [15]. Modern OOP languages such as Scala [34], Ceylon [29], Flow [19] or TypeScript [12] all support the aforementioned type system features.



© Bruno C. d. S. Oliveira, Cui Shaobo, and Baber Rehman;
licensed under Creative Commons License CC-BY

34th European Conference on Object-Oriented Programming (ECOOP 2020).

Editors: Robert Hirschfeld and Tobias Pape; Article No. 29; pp. 29:1–29:29

Leibniz International Proceedings in Informatics



LIPICs Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany



As programming languages evolve, new features are added. This requires that subtyping for these new features is developed and also integrated with existing features. However, the design and implementation of subtyping for new features is quite often non-trivial. There are several, well-documented issues in the literature. These include finding algorithmic forms for subtyping (for instance doing transitivity elimination) [43] or proving metatheoretical properties such as transitivity or narrowing [1]. Such issues occur, for instance, in some of the latest developments for OOP languages, such as the DOT calculi (which model the essence of Scala) [3]. One possible way to reduce the non-trivial amount of work needed to develop new features, would be if two related features could be developed at once with a coherent design. This paper explores a new methodology that enables such benefits.

Normally programming language features are designed independently of each other. However there are features that are closely related to each other, and can be viewed as *dual features*. Various programming language features are known to be dual in programming language theory. For instance sum and product types are well-known to be duals [14]. Similarly universal and existential quantification are dual concepts as well [9]. Moreover duality is a key concept in category theory [30] and many abstractions widely used in functional programming (such as Monads and CoMonads [44]) are also known to be duals.

In OOP type systems dual features are also common. For instance all OOP languages contain a *top type* (called *Object* in Java or *Any* in Scala), which is the supertype of all types. Many OOP languages also contain a *bottom type*, which is a subtype of all types. Top and bottom types can be viewed as dual features, mirroring the functionality of each other. *Intersection* and *union* types are another example of dual features. The intersection of two types A and B can be used to type a value that implements *both A and B*. The union of two types A and B can be used to type a value that implements *either A or B*.

Duality in OOP and subtyping is often only informally observed by humans. For instance, by simply understanding the behaviour of the features and observing their complementary roles, as we just did in the previous paragraph. At best duality is more precisely observed by looking at the rules for the language constructs and their duals and observing a certain symmetry between those rules. However existing formalisms and language designs for type systems and subtyping relations do not directly incorporate duality. Unfortunately this means that an opportunity to exploit obvious similarities between features is lost.

This paper proposes a novel methodology for designing subtyping relations that exploits duality between features directly in the formalism. At the core of our methodology is a generalization of subtyping relations, which we call *Duotyping*. *Duotyping* is parameterized by the mode of the relation. One of these modes is the usual subtyping, while another mode is supertyping (the dual of subtyping). Using the mode it is possible to generalize the usual rules of subtyping to account not only for the intended behaviour of one particular language construct, but also of its dual. This means that the behaviour of the language construct and its dual is modelled by a single, common set of rules. In turn this ensures that the behaviour of the two features is modelled consistently. Moreover it also enables various theorems/properties of subtyping to be generalized to account for the dual features. Therefore, *Duotyping* offers similar benefits to the how duality is exploited in category theory. More concretely, *Duotyping* brings multiple benefits for the design of subtyping relations, which are discussed next.

Shorter specifications. When duality is exploited in specifications of subtyping it leads to shorter specifications because rules for dual features are shared. This also ensures a consistent design of the rules between the dual features directly in the formalism. Such consistency is not

enforceable in traditional formulations of subtyping where the rules are designed separately, and thus their design is completely unconstrained with respect to the dual feature. A concrete example that illustrates shorter specifications is a traditional subtyping relation with top, bottom, union and intersection types, which would normally have 8 subtyping rules for those constructs. In a design with **Duotyping** we only need 5 subtyping rules. Basically we need only *half of the rules* (4 in this case) to model the feature-specific rules, plus an additional *duality rule* which is generic (and plays a similar role to *reflexivity* and *transitivity*).

“Buy” one feature get one feature for free! Duality can lead to the discovery of new features. While top and bottom types, or intersection and union types are well-known in the literature (and understood to be duals), other features in languages with subtyping do not have a known dual feature in the literature. This is partly because, when a language designer employs traditional formulations of subtyping, he/she is often only interested in the design of a feature (but not necessarily of its dual). Even for the case of union and intersection types, intersection types were developed first and the development of union types occurred years later. Because the dual feature is often also useful, the traditional way to design subtyping rules represents a loss of opportunity to get another language feature essentially for free.

One well-known example of a language feature that has been widely exploited in the literature, but its dual feature has received much less attention is *bounded quantification* [17]. Bounded quantification allows type variables to be defined with *upper bounds*. However *lower bounds* are also useful. One can think of universal quantification with lower bounds as a dual to universal quantification with upper bounds. The essence of (upper) bounded quantification is captured by the well-known $F_{<}$ calculus [17]. However, as far as we know, there is no design that extends $F_{<}$ with lower bounded quantification in the literature. Applying a **Duotyping** design to $F_{<}$ gives us, naturally, the two features at once (lower and upper bounded quantification), as illustrated in our Section 4. Such generalization of bounded quantification is related to the recent form of universal quantification with type bounds employed in Scala and the DOT family of calculi [3]. However, while Scala’s type bounds are more expressive than what we propose, they are also much more complex and are in fact one of the key complications in the type systems of languages like Scala. Most DOT calculi require a built-in transitivity rule in subtyping because it is not known how to eliminate transitivity. In contrast, the generalization of $F_{<}$ proposed by us has a formulation of subtyping where transitivity can be proved as a separate lemma.

New proof techniques. Designs of subtyping with duality also enable new proof techniques that exploit such duality. For instance there are various theorems that can be stated for both a feature and its dual, instead of having separate theorems for both. Some of the properties of union and intersection types are examples of this. Moreover, **Duotyping** also enables new proof techniques to prove traditionally hard theorems such as transitivity. Surprisingly to us, for the vast majority of the calculi that we have applied **Duotyping** to, transitivity proofs have been considerably simpler than their corresponding traditional formulations due to the use of **Duotyping**!

Shorter implementations. Finally **Duotyping** also enables for shorter implementations. The benefits of shorter implementations are similar and follow from the benefits of shorter specifications. However there is a complicating factor when moving from a *relational specification* into an implementation: the *duality rule* is non-algorithmic. This is akin to what happens with transitivity, which is often also used in declarative formulations of subtyping.

Eliminating transitivity to obtain an algorithmic system can often be a non-trivial challenge (as illustrated, for instance, by the DOT family of calculi [3]). However, we show that there is a simple and generally applicable technique that can be used to move from a declarative formulation of **Duotyping** into an algorithmic version. This contrasts with transitivity, for which there is not a generally applicable transitivity elimination technique.

To evaluate a design based on **Duotyping** against traditional designs of subtyping, we formalized various calculi with common OOP features (including union types, intersection types and bounded quantification) in Coq in both styles. Our results show that the metatheory when using **Duotyping** has similar complexity and size compared to traditional designs. However, the **Duotyping** formalizations come with more features (for instance lower-bounded quantification) that dualize other well-known features (upper-bounded quantification). Finally, we also show that **Duotyping** can significantly simplify transitivity proofs for many of the calculi studied by us.

In summary, the contributions of this paper are:

- **Duotyping:** A new methodology for the design of subtyping relations exploiting duality.
- **A case study on Duotyping:** A comprehensive study of various existing type systems and features, which were redesigned to employ the **Duotyping** methodology. Our results show that in most systems the size of the metatheory without duality and with duality is comparable, while often transitivity proofs become simpler when employing duality.
- **$F_{<}$: with lower bounded quantification:** We propose a new generalization of System $F_{<}$, called $F_{k\delta}$, which allows not only type variables to be quantified with upper bounds and lower bounds as well. While this system is weaker than Scala/DOT's type bounds, it nonetheless allows for simple transitivity proofs (which have been a significant challenge in calculi with type bounds [40]).
- **Mechanization in Coq:** All the systems in our case study have been formalized in the Coq theorem prover [8].

2 Overview

This section gives an overview of **Duotyping**. We show how to design subtyping relations employing **Duotyping**, and discuss the advantages of a design with **Duotyping** instead of a traditional subtyping formulation in more detail.

2.1 Subtyping with union and intersection types

To motivate the design of **Duotyping** relations we first consider a traditional subtyping relation with union and intersection types, as well as top and bottom types. We choose a system with union and intersection types because these features are nowadays common in various OOP languages, including Scala [34], TypeScript [12], Ceylon [29] or Flow [19]. Therefore union and intersection types are of practical interest. Furthermore union and intersection types are simple, intuitive and good for showing duality between concepts.

The types used for the subtyping relation include the top type \top , the bottom type \perp , integer types Int , function types $A \rightarrow B$, intersection types $A \wedge B$ and union types $A \vee B$:

Types $A, B ::= \top \mid \perp \mid \text{Int} \mid A \rightarrow B \mid A \wedge B \mid A \vee B$

$$\boxed{A <: B} \quad \text{(Traditional Subtyping)}$$

$$\begin{array}{c}
\frac{}{A <: \top} \text{TS-TOP} \quad \frac{}{\perp <: A} \text{TS-BTM} \quad \frac{}{\text{Int} <: \text{Int}} \text{TS-INT} \quad \frac{B_1 <: A_1 \quad A_2 <: B_2}{A_1 \rightarrow A_2 <: B_1 \rightarrow B_2} \text{TS-ARROW} \\
\\
\frac{A <: A_1 \quad A <: A_2}{A <: A_1 \wedge A_2} \text{TS-ANDA} \quad \frac{A_1 <: A}{A_1 \wedge A_2 <: A} \text{TS-ANDB} \quad \frac{A_2 <: A}{A_1 \wedge A_2 <: A} \text{TS-ANDC} \\
\\
\frac{A_1 <: A \quad A_2 <: A}{A_1 \vee A_2 <: A} \text{TS-ORA} \quad \frac{A <: A_1}{A <: A_1 \vee A_2} \text{TS-ORB} \quad \frac{A <: A_2}{A <: A_1 \vee A_2} \text{TS-ORC}
\end{array}$$

■ **Figure 1** Subtyping for union and intersection types.

Traditional Subtyping. A simple subtyping relation accounting for union and intersection types is given in Figure 1. Rule TS-TOP defines that every type is a subtype of \top , and Rule TS-BTM states that every type is a supertype of \perp . Rule TS-INT is for integers, and states that Int is a subtype of itself. Rule TS-ARROW is the traditional subtyping rule for function types. Rules TS-ANDA, TS-ANDB, and TS-ANDC are subtyping rules for intersection types. Rules TS-ORA, TS-ORB, and TS-ORC are subtyping rules for union types. The rules that we employ here are quite common for systems with union and intersection types. For instance they are the same rules used in various DOT-calculi [3] (which model the essence of Scala). For simplicity we do not account for distributivity rules, which also appear in some type systems and calculi [7, 11, 47].

2.2 Subtyping Specifications using Duotyping

In the subtyping relation presented in Figure 1, it is quite obvious that many rules look alike. Some rules are essentially a “mirror image” of other rules. The rules for top and bottom types are an example of this. Another example are the rules TS-ANDB and TS-ORB. Although informally humans can easily observe the similarity between many of the rules, this similarity/duality is not expressed directly in the formalism. For example, there is nothing preventing us from designing rules that are not duals. Duotyping aims at capturing duality in the rules themselves, and expressing duality as part of the formalism, rather than just leaving duality informally observable by humans. This can prevent, for instance, designing rules for dual concepts that do not really dualize. Therefore Duotyping can enforce consistency of dual rule designs.

To illustrate how Duotyping rules are designed and relate to the traditional subtyping rules, let's refactor the traditional rules in a few basic steps. Firstly, let's assume that we have a second relation $A := B$ that captures the *supertyping* between a type A and B . Supertyping is nothing but the subtyping relation with its arguments flipped. So, the rules of supertyping could be simply obtained by taking all the rules in Figure 1 and deriving corresponding rules where all the arguments are flipped around. We skip that boring definition here. With both supertyping and subtyping, the top and bottom rules can be presented as follows:

$$\frac{}{A <: \top} \text{TS-TOP} \quad \frac{}{A := \perp} \text{TSP-BTM}$$

29:6 The Duality of Subtyping

Similarly the rules rule TS-ANDB and rule TS-ORB, can be presented as:

$$\frac{A_1 <: A}{A_1 \wedge A_2 <: A} \text{TS-ANDB} \qquad \frac{A_1 :> A}{A_1 \vee A_2 :> A} \text{TSP-ORB}$$

This simple refactoring shows that the only difference between dual rules is the relation itself, and the (dual) language constructs. Apart from that everything else is the same.

Duotyping. With Duotyping we can provide a single unified rule, which captures the two distinct subtyping rules, instead. The Duotyping relation is parameterized by a mode \diamond :

$$\text{Mode } \diamond ::= <: \mid :>$$

which can be subtyping ($<:$) or supertyping ($:>$). Thus the Duotyping relation is of the form:

$$A \diamond B$$

The mode \diamond is a (third) *parameter* of the relation (besides A and B). With this mode in place, we can readily capture the two refactored rules for supertyping of bottom types and subtyping of top types as two Duotyping rules. However this still requires us to write two distinct rules. To unify those rules into a single one, we introduce a function $\lceil \diamond \rceil$ that chooses the right bound depending on the mode being used:

$$\begin{aligned} \lceil <: \rceil &= \top \\ \lceil :> \rceil &= \perp \end{aligned}$$

If the mode is subtyping the upper bound of the relation is the top type, otherwise it is the bottom type. With $\lceil \diamond \rceil$ we can then write a single unified rule that captures the upper bounds of subtyping and supertyping, and which generalizes both rule TS-TOP and rule TSP-BTM:

$$\frac{}{A \diamond \lceil \diamond \rceil} \text{GDS-TOPBTM}$$

The Duality rule. The Duotyping rule above captures the 2 rules that were refactored above. However there are 4 rules in total for top and bottom types (two for subtyping and two for supertyping). The two missing rules are:

$$\frac{}{\perp <: A} \text{TS-BTM} \qquad \frac{}{\top :> A} \text{TSP-TOP}$$

To capture these missing rules, the Duotyping relation includes a special duality rule:

$$\frac{B \overline{\diamond} A}{A \diamond B} \text{GDS-DUAL}$$

which simply inverts the mode and flips the arguments of the relation. The definition of $\overline{\diamond}$ is, unsurprisingly:

$$\begin{aligned} \overline{<:} &= :> \\ \overline{:>} &= <: \end{aligned}$$

With the duality rule it is clear that the two missing rules are now derivable from the Duotyping rule for bounds and the duality rule. In essence this is the overall idea of the design of Duotyping rules.

$$\boxed{A \diamond B} \qquad \text{(Declarative Duotyping)}$$

$$\begin{array}{c}
\frac{}{A \diamond \top \diamond \perp} \text{GDS-TOPBTM} \qquad \frac{}{\text{Int} \diamond \text{Int}} \text{GDS-INT} \qquad \frac{A_1 \bar{\diamond} A_2 \quad B_1 \diamond B_2}{A_1 \rightarrow B_1 \diamond A_2 \rightarrow B_2} \text{GDS-ARROW} \\
\frac{A \diamond C}{(A \diamond_{?} B) \diamond C} \text{GDS-LEFT} \qquad \frac{B \diamond C}{(A \diamond_{?} B) \diamond C} \text{GDS-RIGHT} \qquad \frac{A \diamond B \quad A \diamond C}{A \diamond (B \diamond_{?} C)} \text{GDS-BOTH} \\
\frac{B \bar{\diamond} A}{A \diamond B} \text{GDS-DUAL}
\end{array}$$

■ **Figure 2** The Duotyping relation for a calculus with union and intersection types.

Complete set of rules. Figure 2 shows the complete version of declarative Duotyping rules for a system with union and intersection types. Rule GDS-TOPBTM defines the rule bounds (which generalizes the rules for top and bottom types). Rule GDS-INT is a simple rule for integers. Int is subtype and supertype of Int. Rule GDS-ARROW is an interesting case. In the first premise $A \bar{\diamond} B$ we invert the mode instead of flipping the arguments of the relation, as done in rule TS-ARROW. One side-effect of this change is that it keeps the rule fully *covariant*, which contrasts with subtyping relations where for arrow types we need contravariance for subtyping of the inputs. This apparently innocent change has important consequences and plays a fundamental role to simplify transitivity proofs as we shall see in Section 2.5.

Rules GDS-LEFT, GDS-RIGHT, and GDS-BOTH each generalize two rules in the traditional formulation of subtyping. Rule GDS-LEFT generalizes rules TS-ANDB and TS-ORB. Rule GDS-RIGHT generalizes rules TS-ANDC and TS-ORC. Rule GDS-BOTH generalizes rules TS-ANDA and TS-ORA. In the three rules an operation $A \diamond_{?} B$ is used:

$$\begin{aligned}
A <_{?} B &= A \wedge B \\
A >_{?} B &= A \vee B
\end{aligned}$$

This operation is used to choose between intersection or union types depending on the mode. If the Duotyping mode is subtyping then we get a rule for intersection types, otherwise we get a dual rule for union types.

Uniform and dual rules. In the context of Duotyping it is useful to distinguish between two different kinds of rules: uniform rules and dual rules.

Uniform rules are those that are essentially the same for supertyping and subtyping. Rules GDS-INT and GDS-ARROW are uniform rules. In those rules the arguments of the relation are exactly the same no matter which mode is being used (subtyping or supertyping).

Dual rules are those that employ dual constructs, like the rules for top and bottom or the rules for union and intersections. Rules GDS-TOPBTM, GDS-LEFT, GDS-RIGHT, and GDS-BOTH are dual rules. The interesting point in these rules is that they use different (dual) constructs depending on the mode. For example, when instantiated with subtyping and supertyping, respectively, the rule GDS-TOPBTM results in:

$$\frac{}{A <: \top} \qquad \frac{}{A >: \perp}$$

2.3 Implementations using Duotyping

Figure 2 showed the declarative `Duotyping` rules for a calculus with union and intersection types. All the rules are syntax directed, except for the duality rule (rule `GDS-DUAL`). This rule flips the mode and arguments to generate a formulation using the dual mode: i.e. it flips subtyping to provide the equivalent supertyping formulation and vice versa. A benefit of using a formulation with the duality rule is that it enables a short specification of `Duotyping`. Unfortunately the duality rule is *not algorithmic*, because the duality rule can always be applied indefinitely. In other words naively translating the rules into an program would easily result in a non-terminating procedure. Therefore to obtain an algorithmic formulation some additional work is needed.

Fortunately, for declarative formulations of `Duotyping`, there is a simple technique that can be used to obtain an algorithmic formulation. A key observation is that `Duotyping` only needs to be flipped (with the duality rule) *at most one time*. Flipping the relation two or more times simply gets us back to the starting point. To capture this idea we can use a (boolean) flag that keeps track of whether the procedure has already employed the duality rule or not.

To make such an idea concrete, Figure 3 shows Haskell code that implements a procedure `duo` for determining `Duotyping` for two types. The code is based on the rules in Figure 2, but it uses a boolean flag to prevent the dual rule (the second to last case in `duo`) from being applied indefinitely. The boolean is true in the initial call or recursive calls to structurally smaller arguments. If the algorithm fails for the first five cases (which are basically a direct translation of the rules `GDS-TOPBTM`, `GDS-INT`, `GDS-ARROW`, `GDS-LEFT`, and `GDS-RIGHT`), then the algorithm simply flips the boolean flag, mode and arguments to run over a dual formulation. This is the second to last line of the algorithm.

For example, if the algorithm is called with the mode set to subtyping and it is not able to find any matching case with the first 5 rules, then it flips the boolean flag to `False`, subtyping to supertyping and the arguments to check the equivalent supertyping formulation. If again it fails to find a matching rule, `False` will be returned and the algorithm will terminate. This illustrates that it is enough to flip the boolean flag once to exploit `Duotyping`. In all our Coq formulations of `Duotyping` we have developed an alternative algorithmic formulation of `Duotyping` which uses an extra boolean flag and is shown to be sound and complete to the declarative formulations with the duality rule. In short there is an easy, general and provably *sound* and *complete* way to implement algorithms based on the idea of `Duotyping`, while at the same time retaining the benefits of reuse of the logic for rules for dual constructs.

2.4 Discovering new features

`Duotyping` can provide interesting extra features essentially for free. For example, the hallmark feature of the well-known $F_{<}$ calculus (a polymorphic calculus with subtyping) [15] is *bounded quantification*, which is a feature used in most modern OOP languages (such as Scala or Java). In $F_{<}$, bounded quantification allows type variables to be defined with *upper bounds*. For example, the following Scala program illustrates the use of such upper bounds:

```
class Person {
  def name: String = "person"
}

class Student extends Person {
  override def name: String = "student"
  def id: String = "id"
}
```



```

data Op = And | Or
data Typ = TInt | TArrow Typ Typ | TOp Op Typ Typ | TBot | TTop
data Mode = Sub | Sup

duo :: Bool -> Mode -> Typ -> Typ -> Bool
duo f m TInt TInt = True
duo f m _ b | b == mode_to_sub m = True
duo f m (TArrow a b) (TArrow c d) =
    duo True (flip m) a c && duo True m b d
duo f m (TOp op a b) c | choose m == op = duo True m a c || duo True m b c
duo f m a (TOp op b c) | choose m == op = duo True m a b && duo True m a c
duo True m a b = duo False (flip m) b a
duo _ _ _ _ = False

```

■ **Figure 3** Haskell code for implementing an algorithmic formulation of Duotyping rules.

```

class StudentsCollection[S <: Student](obj: S) {
  def student: S = obj
}

```

The Scala program shown above uses the upper bounds for the class *StudentsCollection* written as $S <: \text{Student}$. This upper bound restricts *StudentsCollection* to be instantiated with *Student* and its subtypes. Since the upper bound is *Student*, any class that is supertype of *Student* like *Person* cannot be instantiated in *StudentsCollection*.

However *lower bounds* are also useful, and indeed the Scala language allows them (though Java does not). One example of a program with lower bounds in Scala is:

```

class GraduateStudent extends Student {
  def degree: String = "graduate degree"
}

class ResearchStudent extends GraduateStudent {
  override def degree: String = "research degree"
}

class CollectionExcludingResearchStudents[S >: GraduateStudent](obj: S) {
  def student: S = obj
}

```

In contrast to the upper bounds, the Scala program shown above uses the lower bounds for the class *CollectionExcludingResearchStudents* written as $S >: \text{GraduateStudent}$. This lower bound restricts *CollectionExcludingResearchStudents* to be instantiated with *GraduateStudent* and its supertypes. Since the lower bound is *GraduateStudent*. Any class that is a subtype of *GraduateStudent* (such as *ResearchStudent*) cannot be instantiated in *CollectionExcludingResearchStudents*. But any supertype of *GraduateStudent* like *Student* and *Person* (including *GraduateStudent*) can be instantiated in *CollectionExcludingResearchStudents*.

One can think of universal quantification with lower bounds as a dual to universal quantification with upper bounds. While there is no extension of $F_{<}$: that we know of that presents universal quantification with lower bounds in the literature, applying a Duotyping design to $F_{<}$: gives us, naturally, the two features at once (lower and upper bounded quantification).

29:10 The Duality of Subtyping

Bounded quantification in $F_{<}$. The traditional subtyping rule of System kernel $F_{<}$ with upper bounded quantification is:

$$\frac{\Gamma, X <: A \vdash B <: C}{\Gamma \vdash (\forall X <: A. B) <: (\forall X <: A. C)} \text{TS-FORALLKFS}$$

In the premise of this rule, we add the type variable X to the context with an upper bound A . If under the extended context the bodies of the universal quantifier (B and C) are in a subtyping relation then the universal quantifiers are also in a subtyping relation.

To add lower bounded quantification the obvious idea is to add a second rule:

$$\frac{\Gamma, X := A \vdash B <: C}{\Gamma \vdash (\forall X := A. B) <: (\forall X := A. C)} \text{TS-FORALLKFSB}$$

However this alone is not quite right because the environment is also extended with a lower bound ($X >: A$), which does not exist in $F_{<}$ contexts. Therefore some additional care is also needed for the variable cases of $F_{<}$ extended with lower bounded quantification. When an upper bounded constraint is found in the environment, the variable case needs to deal with the upper bound appropriately. Since there are two rules dealing with the variable case in $F_{<}$, one possible approach is to add two more rules for dealing with upper bounds:

$$\frac{X := A \in \Gamma \quad \Gamma \vdash B <: A}{\Gamma \vdash B <: X} \text{TS-TVARB} \qquad \frac{X := A \in \Gamma}{\Gamma \vdash X <: X} \text{TS-REFLTVAR}$$

However such a design feels a little unsatisfactory. We need a total of 6 rules to fully deal with lower and upper bounded quantification (instead of 3 rules in $F_{<}$). At the same time the rules are nearly identical, differing only on the kind of bounds that is used. Furthermore the metatheory of $F_{<}$ also needs to be significantly changed. In particular *narrowing* has to be adapted to account for the lower bounds and *transitivity* has to be extended with several new cases. Since both *narrowing* and *transitivity* proofs for $F_{<}$ are non-trivial, this extension is also non-trivial and adds further complexity to already complex proofs.

A variant of Kernel $F_{<}$ with Duotyping. We now reconsider the design of Kernel $F_{<}$ from scratch employing the **Duotyping** methodology. In the subtyping rule for universal quantification, it is important to note that the subtyping relation between two universal quantifiers in the conclusion is the same as the relation between types B and C in premise. Similarly, the (subtyping/supertyping) bounds of type variable X in the conclusion are the same as the bounds of type variable X in premise. In a design with **Duotyping**, we would like to generalize the two uses of subtyping. Therefore, we can design a single unified rule with the help of two modes:

$$\frac{\Gamma, X \diamond_1 A \vdash B \diamond_2 C}{\Gamma \vdash (\forall X \diamond_1 A. B) \diamond_2 (\forall X \diamond_1 A. C)} \text{GS-FORALLKFS}$$

Section 4.1 explains the **Duotyping** rules of our **Duotyping** kernel $F_{<}$ variant with union and intersection types ($F_k^{\wedge \vee}$) in detail. Rule GS-FORALLKFS is the interesting case, capturing both upper and lower bounded quantification in an elegant way. This rule states that if in a well-formed context, type variable X has a \diamond_1 relation with type A and if type B has \diamond_2

$$\boxed{A \diamond B} \quad (\text{Algorithmic Duotyping})$$

$$\frac{}{A \diamond \top} \text{GS-TOPBTMA} \quad \frac{}{\top \diamond A} \text{GS-TOPBTMB} \quad \frac{}{\text{Int} \diamond \text{Int}} \text{GS-INT}$$

$$\frac{A_1 \overline{\diamond} A_2 \quad B_1 \diamond B_2}{A_1 \rightarrow B_1 \diamond A_2 \rightarrow B_2} \text{GS-ARROW}$$

■ **Figure 4** The Duotyping relation for simply typed lambda calculus.

relation with type C , then the universal quantification with body B has a \diamond_2 relation with the universal quantification with body C . Correspondingly there are also two Duotyping rules for variables:

$$\frac{X \diamond A \in \Gamma \quad \Gamma \vdash A \diamond B}{\Gamma \vdash X \diamond B} \text{GS-TVARA} \quad \frac{X \diamond_1 A \in \Gamma}{\Gamma \vdash X \diamond_2 X} \text{GS-REFLTVAR}$$

In short, the design of a variant of $F_{<}$ with Duotyping leads to a system that naturally accounts for both upper and lower bounded quantification. Moreover, the metatheory, and in particular the proofs of *narrowing* and *transitivity* are not more complex than the corresponding original $F_{<}$ proofs. In fact the proof of transitivity is significantly simpler, because Duotyping enables novel proof techniques as we discuss next.

2.5 New proof techniques

Transitivity proofs are usually a challenge for systems with subtyping. This is partly because subtyping relations often need to deal with some *contravariance*. For instance, the rule TS-ARROW (in Figure 1) is contravariant on the input types. Such contravariance causes problems in certain proofs, including transitivity. To illustrate the issue more concretely, let's distill the essence of the problem by considering a simple lambda calculus with subtyping called $\lambda_{<}$, where the types are:

$$\frac{}{\text{Types} \quad A, B ::= \top \mid \text{Int} \mid A \rightarrow B}$$

and the subtyping rules for those types are just the relevant subset of the rules in Figure 1. The transitivity proof for this simple calculus is:

► **Lemma 1** ($\lambda_{<}$ Transitivity). *If $A <: B$ and $B <: C$ then $A <: C$.*

Proof. By induction on type B .

- Case \top and case Int are trivial to prove by destructing the hypothesis in context.
- Case $B_1 \rightarrow B_2$ requires inversion of the two hypotheses to discover that A can only be a function type, while C is either a function type or \top . ◀

In the arrow case, we need to invert both hypotheses to discover more information about A and C . For this very simple language this double inversion is not too problematic, but as the language of types grows and the subtyping relation becomes more complicated, such inversions become significantly harder to deal with.

At this point one may wonder if the transitivity proof could be done using a different inductive argument to start with, and thus avoid the double inversions. After all there are various other possible choices. Perhaps the most obvious choice is to try induction on the

29:12 The Duality of Subtyping

subtyping relation itself ($A <: B$), rather than on type B . However this does not work because of the contravariance for arrow types, which renders one of induction hypothesis in the arrow case useless (and thus do not allow the case to be proved). Other alternative choices for an inductive argument (such as type A or C) do not work for similar reasons.

Developing metatheory with Duotyping. In order to develop metatheory with Duotyping it is convenient to use an equivalent formulation of Duotyping that eliminates the duality rule (which is non-algorithmic and makes inversions more difficult). For λ_\diamond , which is a Duotyping version of $\lambda_{<}$, this would lead to the set of rules in Figure 4. This alternative algorithmic version eliminates the duality rule. Rules GS-TOPBTMA, GS-INT, and GS-ARROW are similar to the rules we discussed in Section 2.2. Rule GS-TOPBTMB is the dual rule of rule GS-TOPBTMA. With Rule GS-TOPBTMB, the duality rule is unnecessary.

Transitivity with Duotyping. Now we turn our attention to the proof of transitivity:

► **Lemma 2** (λ_\diamond Transitivity). *If $A \diamond B$ and $B \diamond C$ then $A \diamond C$.*

Proof. By induction on $A \diamond B$.

■ All cases are trivial to prove by destructing \diamond and inversion of the second hypothesis ($B \diamond C$). ◀

Transitivity of systems with Duotyping can often be proved by induction on the subtyping relation itself. This has the nice advantage that all the cases essentially become trivial to prove (for λ_\diamond) and only a single inversion is needed for arrow types. A key reason why such approach works in the formulation with Duotyping is that we can keep case for arrow types covariant. Instead we only flip the mode. Another important observation is that when we prove a transitivity lemma with Duotyping we are, in fact, proving two lemmas simultaneously: one lemma for transitivity of subtyping, and another one for transitivity of supertyping. When we use the induction hypothesis we have access to both lemmas (by choosing the appropriate mode).

The proof of the transitivity lemma by induction on the Duotyping relation can scale up to more complex subtyping/Duotyping relations. This includes subtyping relations with advanced features such as intersection types, union types, parametric polymorphism and bounded quantification. All of these can follow the same strategy (induction on the Duotyping relation) to simplify the transitivity proof, as we shall see in Section 5.

3 The $\lambda_\diamond^{\wedge\vee}$ calculus

In Section 2 we gave an overview and discussed advantages of using the Duotyping relation. In this section we introduce a lambda calculus with union and intersection types that is based on Duotyping. We aim at showing that developing calculi and metatheory using Duotyping is simple, requiring only a few small adaptations compared with more traditional formulations based on subtyping. Our main aim is to show type soundness (subject-reduction and preservation) for $\lambda_\diamond^{\wedge\vee}$.

3.1 Syntax and Duotyping

Syntax. Figure 5 shows the syntax of the calculus. The types for $\lambda_\diamond^{\wedge\vee}$ were already introduced in Section 2. Terms include all the constructs for the lambda calculus (variables x , functions $\lambda x : A. e$ and applications $e_1 e_2$) and integers (n). Values are a subset of terms, consisting of

Types	A, B	$::=$	$\top \mid \perp \mid \text{Int} \mid A \rightarrow B \mid A \wedge B \mid A \vee B$
Terms	e	$::=$	$x \mid n \mid \lambda x : A. e \mid e_1 e_2$
Values	v	$::=$	$n \mid \lambda x : A. e$
Context	Γ	$::=$	$\bullet \mid \Gamma, x : A$
Mode	\diamond	$::=$	$<: \mid :>$

 $A \diamond B$ *(Algorithmic Duotyping)*

$$\begin{array}{c}
\frac{A \diamond C}{(A \diamond_? B) \diamond C} \text{GS-LEFTA} \qquad \frac{A \diamond B}{A \diamond (B \overline{\diamond}_? C)} \text{GS-LEFTB} \qquad \frac{B \diamond C}{(A \diamond_? B) \diamond C} \text{GS-RIGHTA} \\
\frac{A \diamond C}{A \diamond (B \overline{\diamond}_? C)} \text{GS-RIGHTB} \qquad \frac{A \diamond B \quad A \diamond C}{A \diamond (B \diamond_? C)} \text{GS-BOTHA} \qquad \frac{A \diamond C \quad B \diamond C}{(A \overline{\diamond}_? B) \diamond C} \text{GS-BOTHB}
\end{array}$$

■ **Figure 5** Syntax and Duotyping relation for union and intersection types.

abstractions and integers only. The mode \diamond is used to choose the mode of the relation: it can be either subtyping ($<:$) or supertyping ($:>$). Typing contexts Γ are standard and used to track the types of the variables in a program. Finally, a well-formedness relation $\Gamma \vdash \mathbf{ok}$ ensures that typing contexts are well-formed.

Duotyping for $\lambda_{\diamond}^{\wedge \vee}$. The Duotyping rules for $\lambda_{\diamond}^{\wedge \vee}$ were already partly presented in Figure 2. In addition to the rules in the λ_{\diamond} , we also need extra rules for union and intersection types. These extra rules are presented in Figure 5. Rules GS-LEFTA, GS-RIGHTA, and GS-BOTHA are also similar to the rules GDS-LEFT, GDS-RIGHT, and GDS-BOTH presented in Figure 2. Since we eliminate the *duality rule* in the algorithmic version, we add dual subtyping rules. Rules GS-LEFTB, GS-RIGHTB, and GS-BOTHB are the dual versions of rules GS-LEFTA, GS-RIGHTA, and GS-BOTHA respectively. This formulation is shown to be sound and complete with respect to the formulation with the duality rule in Figure 2. As explained in Section 2 this variant of the rules makes some proofs easier, thus we employ it here. The Duotyping relation is reflexive and transitive:

► **Theorem 3** (Reflexivity). $A \diamond A$.

Proof. By induction on type A . Reflexivity is trivial to prove by applying subtyping rules. ◀

► **Theorem 4** (Transitivity). *If $A \diamond B$ and $B \diamond C$ then $A \diamond C$.*

Proof. By induction on subtyping relation.

- Cases rule GS-TOPBTMA, rule GS-INT, rule GS-LEFTA, rule GS-RIGHTA and rule GS-BOTHA are trivial to prove.
- Case rule GS-TOPBTMB requires an additional Lemma 5.
- Case rule GS-ARROW requires induction on hypothesis and subtyping rules.
- Cases rule GS-LEFTB and rule GS-RIGHTB requires an additional Lemma 6 to be applied on hypothesis in context.
- Case rule GS-BOTHB requires induction on the hypothesis. This case also requires rule GS-LEFTB, rule GS-RIGHTB, and rule GS-BOTHA subtyping rules. ◀

We used the following auxiliary lemmas to prove transitivity.

29:14 The Duality of Subtyping

$$\boxed{\Gamma \vdash e : A} \quad (\text{Typing})$$

$$\frac{\Gamma \vdash \mathbf{ok} \quad x : A \in \Gamma}{\Gamma \vdash x : A} \text{G-VAR} \quad \frac{\Gamma \vdash \mathbf{ok}}{\Gamma \vdash n : \mathbf{Int}} \text{G-INT} \quad \frac{\Gamma, x : A_1 \vdash e_2 : A_2}{\Gamma \vdash \lambda x : A_1. e_2 : A_1 \rightarrow A_2} \text{G-ABS}$$

$$\frac{\Gamma \vdash e_1 : A_1 \rightarrow A_2 \quad \Gamma \vdash e_2 : A_1}{\Gamma \vdash e_1 e_2 : A_2} \text{G-APP} \quad \frac{\Gamma \vdash e : B \quad B <: A}{\Gamma \vdash e : A} \text{G-SUB}$$

$$\boxed{e_1 \longrightarrow e_2} \quad (\text{Reduction})$$

$$\frac{}{(\lambda x : A_1. e_1) v_2 \longrightarrow [x \mapsto v_2]e_1} \text{GRED-APPABS} \quad \frac{e_1 \longrightarrow e'_1}{e_1 e \longrightarrow e'_1 e} \text{GRED-FUN} \quad \frac{e_1 \longrightarrow e'_1}{v e_1 \longrightarrow v e'_1} \text{GRED-ARG}$$

■ **Figure 6** Typing and reduction for $\lambda_{\diamond}^{\wedge \vee}$.

► **Lemma 5** (Bound Selection). *If $\top \diamond [\diamond B$ then $A \diamond B$.*

This lemma captures the upper and lower bounds with respect to relation between two types. If the mode is subtyping, then it states that any type that is supertype of \top is supertype of all the other types. If the mode is supertyping, then it states that any type that is subtype of \perp is subtype of all the other types. In essence the lemma generalizes the following two lemmas (defined directly over subtyping and supertyping):

- If $\top <: B$ then $A <: B$
- If $\perp >: B$ then $A >: B$

► **Lemma 6** (Inversion for rule GDS-Both). *If $C \diamond (A \diamond ? B)$ then $(C \diamond A)$ and $(C \diamond B)$.*

This lemma captures the relation between types with respect to the duality of union and intersection types. It is the general form of two lemmas:

- **Lemma 7** (Inversion for Union types). *If $(A \vee B) <: C$ then $(A <: C)$ and $(B <: C)$.*
- **Lemma 8** (Inversion for Intersection types). *If $C <: (A \wedge B)$ then $(C <: A)$ and $(C <: B)$.*

Finally there is also a *duality lemma*, which complements reflexivity and transitivity:

► **Lemma 9** (Duality). $A \diamond B = B \bar{\diamond} A$.

This lemma captures the essence of duality, and enables us to switch the mode of the relation by flipping the arguments as well. Furthermore, the duality lemma plays a crucial role when proving soundness and completeness with respect to the declarative version of Duotyping, which has duality as an axiom instead. All of these lemmas are used in later proofs for type soundness.

3.2 Semantics and type soundness

Typing. The first part of Figure 6 presents the typing rules of $\lambda_{\diamond}^{\wedge \vee}$. The rules are standard. Note that rule G-SUB is the subsumption rule: if an expression e has type B and B is a subtype of A then e has type A . Noteworthy, $B <: A$ is the Duotyping relation being used with the subtyping mode.

Reduction. At the bottom of Figure 6 we show the reduction rules of $\lambda_{\diamond}^{\wedge\vee}$. Again, the reduction rules are standard. Rule GRED-APPABS is the usual beta-reduction rule, which substitutes a value v_2 for x in the lambda body e_1 . Rule GRED-FUN and rule GRED-ARG are the standard call-by-value rules for applications.

Type soundness. The proof for type soundness relies on the usual preservation and progress lemmas:

► **Lemma 10** (Type Preservation). *If $\Gamma \vdash e : A$ and $e \longrightarrow e'$ then: $\Gamma \vdash e' : A$.*

Proof. By induction on the typing relation and with the help of Lemma 9. ◀

► **Lemma 11** (Progress). *If $\Gamma \vdash e : A$ then:*

1. *either e is a value.*
2. *or e can take a step to e' .*

Proof. By induction on the typing relation. ◀

3.3 Summary and Comparison

Besides $\lambda_{\diamond}^{\wedge\vee}$, which employs the **Duotyping** relation, we have also formalized a lambda calculus with union and intersection types using the traditional subtyping relation ($\lambda_{<}^{\wedge\vee}$). Most of the metatheory is similar with a great deal of theorems being almost the same. The main differences are in the metatheory for subtyping which has to be generalized. For example both reflexivity and transitivity have to be generalized to operate in the **Duotyping** relation instead. The formalization with **Duotyping** only has two additional lemmas (the duality lemma and the bound selection lemma), which have no counterparts with subtyping. The number of lines of code for the formalization of $\lambda_{<}^{\wedge\vee}$ is 596 whereas for $\lambda_{\diamond}^{\wedge\vee}$ is 630. The total number of lemmas required for $\lambda_{<}^{\wedge\vee}$ are 23 and 25 for $\lambda_{\diamond}^{\wedge\vee}$. Following two lemmas in $\lambda_{<}^{\wedge\vee}$ are captured as one lemma in $\lambda_{\diamond}^{\wedge\vee}$ (Lemma 6):

Inversion for Union Types. This lemma is already stated as Lemma 7: it is the inversion of the subtyping rule for the union types in the traditional subtyping relation. The lemma states that if the union of two types A and B is the subtype of a type C , then both types A and B are subtypes of type C .

Inversion for Intersection Types. This lemma, which corresponds to Lemma 8, is the inversion of the subtyping rule for the intersection types with the traditional subtyping relation. It states that if a type C is the subtype of the intersection of two types A and B , then the type C is a subtype of both types A and B .

4 The $F_{k\diamond}^{\wedge\vee}$ calculus

In Section 3 we introduced a simple calculus with union and intersection types using **Duotyping**. This section extends that calculus with bounded quantification based on kernel $F_{<}$. This new variant also employs **Duotyping** and is called $F_{k\diamond}^{\wedge\vee}$. The main aim of this section is to show that sometimes we can get interesting and novel dual features come for free. In addition to upper bounded quantification of $F_{<}$, System $F_{k\diamond}^{\wedge\vee}$ provides lower bounded quantification as well. Additionally, we also show the type soundness of $F_{k\diamond}^{\wedge\vee}$.

29:16 The Duality of Subtyping

Types	A, B	$::=$	$\top \mid \perp \mid \text{Int} \mid A \rightarrow B \mid A \wedge B \mid A \vee B \mid X \mid \forall(X \diamond A).B$
Terms	e	$::=$	$x \mid n \mid \lambda x : A. e \mid e_1 e_2 \mid \Lambda(X \diamond A).e \mid e A$
Values	v	$::=$	$n \mid \lambda x : A. e \mid \Lambda(X \diamond A).e$
Context	Γ	$::=$	$\bullet \mid \Gamma, x : A \mid \Gamma, X \diamond A$
Mode	\diamond	$::=$	$< : \mid : >$

$\Gamma \vdash A \diamond B$	$(F_{k\diamond}^{\wedge\vee} \text{ Duotyping})$
$\frac{\Gamma \vdash \text{ok} \quad X \diamond_1 A \in \Gamma}{\Gamma \vdash X \diamond_2 X} \text{GS-REFLTVARA}$	$\frac{X \diamond A \in \Gamma \quad \Gamma \vdash A \diamond B}{\Gamma \vdash X \diamond B} \text{GS-TVARA}$
$\frac{X \diamond A \in \Gamma \quad \Gamma \vdash B \overline{\diamond} A}{\Gamma \vdash B \overline{\diamond} X} \text{GS-TVARB}$	$\frac{\Gamma, X \diamond_1 A \vdash B \diamond_2 C}{\Gamma \vdash (\forall X \diamond_1 A. B) \diamond_2 (\forall X \diamond_1 A. C)} \text{GS-FORALLKFS}$

■ **Figure 7** Syntax and additional rules for Duotyping in $F_{k\diamond}^{\wedge\vee}$.

4.1 Syntax and Duotyping

Syntax. Figure 7 shows the syntax of the calculus $F_{k\diamond}^{\wedge\vee}$. Types \top , \perp , Int , $A \rightarrow B$, $A \wedge B$, $A \vee B$ are already introduced in Section 2. Type variable X and a universal quantifier on type variables $\forall(X \diamond A).B$ are the two additional types in $F_{k\diamond}^{\wedge\vee}$. Terms x , n , $\lambda x : A. e$, $e_1 e_2$ are already discussed in Section 3.1. Type abstraction $\Lambda(X \diamond A).e$ and type application $e A$ are two additional terms in $F_{k\diamond}^{\wedge\vee}$. Values are a subset of terms, consisting of term abstraction, type abstraction and integers.

Duotyping for $F_{k\diamond}^{\wedge\vee}$. Duotyping rules for a calculus with union and intersection types are presented in Figure 4. $F_{k\diamond}^{\wedge\vee}$ has two significant differences in its Duotyping rules in comparison to Figure 4, which are presented in Figure 7. The first one is the addition of a typing context in the Duotyping rules. This is important to ensure that type variables are bound. Thus, Duotyping for $F_{k\diamond}^{\wedge\vee}$ is now of the form $\Gamma \vdash A \diamond B$. The second difference is that there are four more rules, three of them (rules GS-REFLTVARA, GS-TVARA, and GS-FORALLKFS) were already explained in Section 2.4. Rule GS-TVARB is the dual of rule GS-TVARA. We introduce this rule to eliminate the *duality rule*.

The Duotyping relation for $F_{k\diamond}^{\wedge\vee}$ is reflexive and transitive as well:

► **Theorem 12** (Reflexivity). $\Gamma \vdash A \diamond A$.

Proof. By induction on type A . ◀

► **Theorem 13** (Transitivity). If $\Gamma \vdash A \diamond B$ and $\Gamma \vdash B \diamond C$ then $\Gamma \vdash A \diamond C$.

Proof. By induction on $\Gamma \vdash A \diamond B$.

- Cases rule GS-TOPBTMA, rule GS-TOPBTMB, rule GS-INT, rule GS-REFLTVAR, rule GS-TVARA, rule GS-LEFTA, rule GS-RIGHTA, rule GS-BOTHA are trivial to prove.
- Case rule GS-ARROW is proved using the induction hypotheses.
- Case rule GS-TVARB can be proved using Lemma 16.
- Case rule GS-FORALLKFS is proved using the induction hypotheses.
- Case rule GS-LEFTB can be proved using an additional Lemma 15.
- Case rule GS-RIGHTB also uses Lemma 15.
- Case rule GS-BOTHB is proved using the induction hypotheses. ◀

The auxiliary lemmas for transitivity are described next and are essentially the same as in Section 3.1.

► **Lemma 14** (Bound Selection). *If $\Gamma \vdash \lceil \diamond \lceil \diamond B$ then $\Gamma \vdash A \diamond B$.*

► **Lemma 15** (Inversion for rule GDS-Both). *If $\Gamma \vdash C \diamond (A \diamond_? B)$ then $\Gamma \vdash (C \diamond A)$ and $(C \diamond B)$.*

There is also a *duality lemma*:

► **Lemma 16** (Duality). $\Gamma \vdash A \diamond B = \Gamma \vdash B \overline{\diamond} A$.

Finally, We also proved weakening and the narrowing lemmas for **Duotyping** calculus. Here we briefly compare the narrowing lemma for $F_{k<}^{\wedge \vee}$ and $F_{k\diamond}^{\wedge \vee}$:

► **Lemma 17** ($F_{k<}^{\wedge \vee}$ Narrowing Lemma). *If $\Gamma \vdash A <: B$ and $\Gamma, X <: B, \Gamma_1 \vdash C <: D$ then $\Gamma, X <: A, \Gamma_1 \vdash C <: D$*

► **Lemma 18** ($F_{k\diamond}^{\wedge \vee}$ Narrowing Lemma). *If $\Gamma \vdash A \diamond_1 B$ and $\Gamma, X \diamond_1 B, \Gamma_1 \vdash C \diamond_2 D$ then $\Gamma, X \diamond_1 A, \Gamma_1 \vdash C \diamond_2 D$*

Lemma 17 exploits only the subtyping relation while Lemma 18 exploits our **Duotyping** relation. Lemma 18 illustrates how lower and upper bounds are captured under a unified mode relation in narrowing. Like the transitivity statement using a **Duotyping** formulation, one can think of the **Duotyping** narrowing lemma as actually two distinct lemmas: one for narrowing of upper bounds and another for narrowing of lower bounds. Also, it is important to note that Lemma 18 is using two modes \diamond_1 and \diamond_2 . \diamond_1 is the relation between types A , B and the type variable X . Whereas, \diamond_2 is the relation between type C and type D . Those two relations do not need to be the same.

4.2 Semantics and type soundness

Typing. The first part of Figure 8 presents the typing rules of $F_{k\diamond}^{\wedge \vee}$. The first five rules are standard and are already explained in Section 2.1. Rules G-TABS and G-TAPP are the two additional rules in $F_{k\diamond}^{\wedge \vee}$. Rule G-TABS is similar to the standard rule for type abstractions in $F_{k<}^{\wedge \vee}$: except that it generalizes the subtyping bound to a \diamond bound, which could either be subtyping or supertyping. Rule G-APP again differs from the rule for type applications in $F_{k<}^{\wedge \vee}$: by using a \diamond bound instead of just a subtyping bound. These two rules are noteworthy because they also illustrate an advantage of using **Duotyping** in the typing relation. Without **Duotyping** we would need multiple typing rules to capture different variations of the bounds.

Reduction. The last part of Figure 8 presents the reduction rules of our calculus. Again, reduction rules are standard except for the rule GRED-TAPPTABS. In rule GRED-TAPPTABS the duality relation captures both upper and the lower bounds. Rule GRED-TFUN is the standard reduction rule for the type applications.

Type Soundness. We proved the type soundness for our calculus. All the proofs are formalized in Coq theorem prover.

► **Lemma 19** (Type Preservation). *If $\Gamma \vdash e : A$ and $e \longrightarrow e'$ then: $\Gamma \vdash e' : A$.*

Proof. By induction on the typing relation.

- Case rules G-VAR, G-INT, G-ABS, G-TABS, and G-SUBS are trivial to solve.
- Case rule G-APP uses Theorem 12 and Lemma 16.
- Case rule G-TAPP uses Theorem 12. ◀

29:18 The Duality of Subtyping

$$\boxed{\Gamma \vdash e : A} \quad (\text{Typing})$$

$$\frac{\Gamma \vdash \text{ok} \quad x : A \in \Gamma}{\Gamma \vdash x : A} \text{G-VAR} \quad \frac{\Gamma \vdash \text{ok}}{\Gamma \vdash n : \text{Int}} \text{G-INT} \quad \frac{\Gamma, x : A_1 \vdash e_2 : A_2}{\Gamma \vdash \lambda x : A_1. e_2 : A_1 \rightarrow A_2} \text{G-ABS}$$

$$\frac{\Gamma \vdash e_1 : A_1 \rightarrow A_2 \quad \Gamma \vdash e_2 : A_1}{\Gamma \vdash e_1 e_2 : A_2} \text{G-APP} \quad \frac{\Gamma \vdash e : B \quad \Gamma \vdash B <: A}{\Gamma \vdash e : A} \text{G-SUBS}$$

$$\frac{\Gamma, X \diamond A \vdash e : B}{\Gamma \vdash \Lambda X \diamond A. e : \forall (X \diamond A). B} \text{G-TABS} \quad \frac{\Gamma \vdash e : \forall (X \diamond A). B \quad \Gamma \vdash C \diamond A}{\Gamma \vdash e C : [X \mapsto C] B} \text{G-TAPP}$$

$$\boxed{e_1 \longrightarrow e_2} \quad (\text{Reduction})$$

$$\frac{}{(\lambda x : A_1. e_1) v_2 \longrightarrow [x \mapsto v_2] e_1} \text{GRED-APPABS} \quad \frac{e_1 \longrightarrow e'_1}{e_1 e \longrightarrow e'_1 e} \text{GRED-FUN} \quad \frac{e_1 \longrightarrow e'_1}{v e_1 \longrightarrow v e'_1} \text{GRED-ARG}$$

$$\frac{}{(\Lambda X \diamond A. e_1) B \longrightarrow [X \mapsto B] e_1} \text{GRED-TAPPABS} \quad \frac{e_1 \longrightarrow e'_1}{e_1 A \longrightarrow e'_1 A} \text{GRED-TFUN}$$

■ **Figure 8** Typing and reduction of the duotyped kernel $F_{<}$.

► **Lemma 20** (Progress). *If $\Gamma \vdash e : A$ then:*

1. *either e is value.*
2. *or e can take step to e' .*

Proof. By induction on the typing relation.

- Case rules G-VAR, G-INT, G-ABS, G-TABS, and G-SUBS are trivial to solve.
- Case rule G-APP requires canonical forms.
- Case rule G-TAPP requires canonical forms. ◀

4.3 Summary and Comparison

Besides $F_{k\diamond}^{\wedge\vee}$, which employs the Duotyping relation, we have also formalized a calculus $F_{k<}^{\wedge\vee}$: an extension of kernel $F_{<}$: (only with upper bounded quantification) with union and intersection types using the traditional subtyping relation. The essential differences are similar to what we already discussed in Section 3.3. The formalization with Duotyping only has two additional lemmas (the duality lemma and the bound selection lemma), besides a few minor auxiliary lemmas. The number of lines for proof for the formalization of $F_{k<}^{\wedge\vee}$ is 1648 whereas for $F_{k\diamond}^{\wedge\vee}$ is 1770. The total lemmas required for $F_{k<}^{\wedge\vee}$ are 74 and 81 for $F_{k\diamond}^{\wedge\vee}$. We emphasize that one significant difference between $F_{k<}^{\wedge\vee}$ and $F_{k\diamond}^{\wedge\vee}$ is the additional lower bounded quantification provided by $F_{k\diamond}^{\wedge\vee}$. This is an extra feature which comes essentially for free with Duotyping.

5 A Case Study on Duotyping

In this section we present an empirical case study, which we conducted to validate some of the benefits of Duotyping. Overall, the results of our case study indicate that: Duotyping does allow for compact specifications; the complexity of developing formalization with Duotyping is comparable to similar developments using traditional subtyping relations; transitivity proofs are often significantly simpler; and Duotyping is a generally applicable technique.

5.1 Case Study

We formalized a number of different calculi using **Duotyping**. All the proofs and metatheory are mechanically checked by the Coq theorem prover. We also formalized a few traditional subtyping systems for comparison. Table 1 shows a brief overview of various systems that we formalized. $\lambda_{<}$, $\lambda_{<}^{\wedge\vee}$, $F_{k<}$, $F_{k<}^{\wedge\vee}$ and $F_{F<}$ are the traditional subtyping systems. The Coq formalizations for the traditional subtyping systems are based on existing Coq formalizations from the locally nameless representation with cofinite quantification tutorial and repository (<https://www.chargueraud.org/softs/ln/>) by Charguéraud [18]. The formalizations of λ_{\diamond} , $\lambda_{\diamond}^{\wedge\vee}$, $F_{k\diamond}$, $F_{k\diamond}^{\wedge\vee}$ and $F_{F\diamond}$ are their respective **Duotyping** formulations, and modify the original ones with traditional subtyping. Subscript $<$ represents a calculus with traditional subtyping whereas \diamond represents a calculus with **Duotyping**. Superscript $\wedge\vee$ is the notation for a system with intersection and union types. Subscript k corresponds to the kernel version of a variant of $F_{<}$, while subscript F corresponds to the corresponding full version. We also formalized a simple polymorphic system without bounded quantification using **Duotyping**. We have two **Duotyping** variants for this polymorphic type system without bounded quantification. One without union and intersection types (F_{\diamond}) and another with union and intersection types ($F_{\diamond}^{\wedge\vee}$).

In Table 1, the last column (Transitivity) summarizes the proof technique used in each system to prove transitivity. Recall the transitivity lemma (using the **Duotyping** formulation):

► **Theorem 21** (Transitivity). *If $A \diamond B$ and $B \diamond C$ then $A \diamond C$.*

Induction on the middle type means induction on type B (or well-formed type B for polymorphic systems), whereas induction on the **Duotyping** relation means induction on $A \diamond B$.

Research Questions. Section 1 discussed benefits of using **Duotyping**. This section attempts to quantify some of these benefits. More concretely, we answer the following questions in this section:

- Does **Duotyping** provide shorter specifications?
- Does **Duotyping** increase the complexity of the formalization and metatheory of the language?
- Does **Duotyping** make transitivity proofs simpler?
- Is **Duotyping** a generally applicable technique?

We follow an empirical approach to answer these questions and address each question in a separate (sub)section. Obviously a precise measure for complexity/simplicity is hard to obtain. We use SLOC for the formalization and proofs as an approximation. All the formalizations are written in the same Coq style to ensure that the comparisons are fair.

5.2 Does **Duotyping** provide shorter specifications?

This section answers our first question. In short our case study seems to support this conclusion. The declarative **Duotyping** rules of all the systems that we formalized are shown in Table 2. Please note that the formulation also contains the *duality rule*. λ_{\diamond} has the basic set of **Duotyping** rules. These rules are common in all of the systems. $\lambda_{\diamond}^{\wedge\vee}$ has the subtyping rules for intersection types and union types in addition to the rules from λ_{\diamond} . F_{\diamond} contains two more rules (rules GDS-REFLTVARP and GDS-FORALLFSP) in addition to the rules from λ_{\diamond} . $F_{\diamond}^{\wedge\vee}$ has all the rules from λ_{\diamond} , $\lambda_{\diamond}^{\wedge\vee}$ and F_{\diamond} . $F_{k\diamond}$ has three additional

■ **Table 1** Description of all systems.

Name	Description	SLOC	Transitivity
$\lambda_{<}$	STLC with subtyping	537	By induction on the middle type.
λ_{\diamond}	STLC with Duotyping	583	By induction on the Duotyping relation.
$\lambda_{<}^{\wedge\vee}$	STLC with subtyping, union types and intersection types	595	By induction on the middle type.
$\lambda_{\diamond}^{\wedge\vee}$	STLC with Duotyping, union types and intersection types	623	By induction on the Duotyping relation.
F_{\diamond}	Simple polymorphic system with Duotyping and without bounded quantification	1466	By induction on the Duotyping relation.
$F_{\diamond}^{\wedge\vee}$	Simple polymorphic system with Duotyping, union types and intersection types and without bounded quantification	1546	By induction on the Duotyping relation.
$F_{k<}$	System $F_{<}$: kernel	1542	By induction on the (well-formed) middle type.
$F_{k\diamond}$	System $F_{<}$: kernel with Duotyping	1579	By induction on the Duotyping relation.
$F_{k<}^{\wedge\vee}$	System $F_{<}$: kernel with subtyping, union types and intersection types	1648	By induction on the (well-formed) middle type.
$F_{k\diamond}^{\wedge\vee}$	System $F_{<}$: kernel with Duotyping, union types and intersection types	1770	By induction on the Duotyping relation.
$F_{F<}$	System full $F_{<}$:	1518	By induction on the (well-formed) middle type.
$F_{F\diamond}$	System full $F_{<}$: with Duotyping	1786	By induction on the (well-formed) middle type.

subtyping rules GDS-REFLTVAR, GDS-TVAR, and GDS-FORALLKFS in addition to the rules from λ_{\diamond} . $F_{k\diamond}^{\wedge\vee}$ has all the rules from λ_{\diamond} , $\lambda_{\diamond}^{\wedge\vee}$, and $F_{k\diamond}$. $F_{F\diamond}$ has an additional subtyping rule GDS-FORALLFFS.

Comparison with systems using traditional subtyping. Table 3 shows the number of rules and features for different calculi formulated with subtyping and Duotyping. In our formulation, $\lambda_{<}$ has 3 types \top , Int , and $A \rightarrow B$. This requires 3 subtyping rules to capture the subtyping relation of these 3 types. If we wanted to support the \perp type in $\lambda_{<}$ we would need to add 1 more subtyping rule. In the table we express the extra rules required for extra features as $(+n)$, where n is the number of extra rules. Duotyping supports \perp for free by exploiting the dual nature of \top with the help of *duality rule*. Systems with more rules follow the same approach for traditional systems i.e more types require more subtyping rules. If we wanted to support the \perp type in $\lambda_{<}^{\wedge\vee}$ we also need 1 additional rule. To further extend our discussion to the polymorphic systems with bounded quantification, we would need 4 additional rules in $F_{k<}$ (1 for \perp type and 3 for lower bounded quantification). Similarly we would need 4 additional rules to support lower bounds and lower bounded quantification in $F_{k<}^{\wedge\vee}$.

In summary, in the systems that we compared Duotyping has a similar number of rules to systems with subtyping, but it comes with extra features. If we wanted to add those features to systems with traditional subtyping, then that would generally result in more rules for the traditional versions compared to Duotyping. This would also have an impact in the SLOC of the metatheory, increasing the metatheory for those systems considerably.

■ **Table 2** Declarative Duotyping rules of all systems.

Name	Duotyping Rules
λ_{\diamond}	$\boxed{A \diamond B} \quad (\lambda_{\diamond} \text{ Duotyping})$ $\frac{}{A \diamond \int \int} \text{GDS-TOPBTM} \quad \frac{}{\text{Int} \diamond \text{Int}} \text{GDS-INT} \quad \frac{A_1 \bar{\diamond} A_2 \quad B_1 \diamond B_2}{A_1 \rightarrow B_1 \diamond A_2 \rightarrow B_2} \text{GDS-ARROW}$ $\frac{B \bar{\diamond} A}{A \diamond B} \text{GDS-DUAL}$
$\lambda_{\diamond}^{\wedge \vee}$	$\boxed{A \diamond B} \quad (\lambda_{\diamond}^{\wedge \vee} \text{ Duotyping plus all rules from } \lambda_{\diamond})$ $\frac{A \diamond C}{(A \diamond ? B) \diamond C} \text{GDS-LEFT} \quad \frac{B \diamond C}{(A \diamond ? B) \diamond C} \text{GDS-RIGHT} \quad \frac{A \diamond B \quad A \diamond C}{A \diamond (B \diamond ? C)} \text{GDS-BOTH}$
F_{\diamond}	$\boxed{A \diamond B} \quad (F_{\diamond} \text{ Duotyping plus all rules from } \lambda_{\diamond})$ $\frac{}{X \diamond X} \text{GDS-REFLTVARP} \quad \frac{A \diamond B}{(\forall X.A) \diamond (\forall X.B)} \text{GDS-FORALLFSP}$
$F_{\diamond}^{\wedge \vee}$	$\boxed{A \diamond B} \quad (F_{\diamond}^{\wedge \vee} \text{ Duotyping plus all rules from } \lambda_{\diamond}, \lambda_{\diamond}^{\wedge \vee} \text{ and } F_{\diamond})$
$F_{k\diamond}$	$\boxed{\Gamma \vdash A \diamond B} \quad (F_{k\diamond} \text{ Duotyping plus all rules from } \lambda_{\diamond})$ $\frac{\Gamma \vdash \text{ok} \quad X \diamond_1 A \in \Gamma}{\Gamma \vdash X \diamond_2 X} \text{GDS-REFLTVAR} \quad \frac{X \diamond A \in \Gamma \quad \Gamma \vdash A \diamond B}{\Gamma \vdash X \diamond B} \text{GDS-TVAR}$ $\frac{\Gamma, X \diamond_1 A \vdash B \diamond_2 C}{\Gamma \vdash (\forall X \diamond_1 A.B) \diamond_2 (\forall X \diamond_1 A.C)} \text{GDS-FORALLKFS}$
$F_{k\diamond}^{\wedge \vee}$	$\boxed{\Gamma \vdash A \diamond B} \quad (F_{k\diamond}^{\wedge \vee} \text{ Duotyping plus all rules from } \lambda_{\diamond}, \lambda_{\diamond}^{\wedge \vee} \text{ and } F_{k\diamond})$
$F_{F\diamond}$	$\boxed{\Gamma \vdash A \diamond B} \quad (F_{F\diamond} \text{ Duotyping plus all rules from } F_{k\diamond} \text{ excluding rule GS-FORALLKFS and union/intersection rules})$ $\frac{\Gamma \vdash A \diamond_1 B \quad \Gamma, X \diamond_1 (A \bar{\diamond} B) \vdash A_1 \diamond_2 B_1}{\Gamma \vdash (\forall X \diamond_1 A.A_1) \diamond_2 (\forall X \diamond_1 B.B_1)} \text{GDS-FORALLFFS}$

29:22 The Duality of Subtyping

■ **Table 3** Comparing the features and number of rules with subtyping and Duotyping.

System	Subtyping rules count	System	Duotyping rules count	Duotyping extra features
$\lambda_{<}$	3 (+1)	λ_{\diamond}	4	lower bounds in λ_{\diamond}
$\lambda_{<}^{\wedge\vee}$	9 (+1)	$\lambda_{\diamond}^{\wedge\vee}$	7	lower bounds in $\lambda_{\diamond}^{\wedge\vee}$
$F_{k<}$	5 (+4)	$F_{k\diamond}$	7	lower bounds and lower bounded quantification in $F_{k\diamond}$
$F_{k<}^{\wedge\vee}$	11 (+4)	$F_{k\diamond}^{\wedge\vee}$	10	lower bounds and lower bounded quantification in $F_{k\diamond}^{\wedge\vee}$

■ **Table 4** SLOC of traditional subtyping and Duotyping systems.

Subtyping System	SLOC	Duotyping System	SLOC
$\lambda_{<}$	537	λ_{\diamond}	583
$\lambda_{<}^{\wedge\vee}$	595	$\lambda_{\diamond}^{\wedge\vee}$	623
$F_{k<}$	1542	$F_{k\diamond}$	1579
$F_{k<}^{\wedge\vee}$	1648	$F_{k\diamond}^{\wedge\vee}$	1770

5.3 Does Duotyping increase the complexity of the formalization and metatheory of the language?

At first, one may think that Duotyping increases the complexity of formalization and metatheory of the language, since it provides interesting extra features and generalizations normally come at a cost. Interestingly, Duotyping does not add significant extra complexity in the formalization and metatheory of the language. Table 4 shows the SLOC for formalizations using traditional subtyping and Duotyping systems. The lines of code for $\lambda_{<}^{\wedge\vee}$ are 595 and the lines of code for $\lambda_{\diamond}^{\wedge\vee}$ are 623. Similarly, the lines of code for $F_{k<}^{\wedge\vee}$ are 1648 and 1770 for $F_{k\diamond}^{\wedge\vee}$. Although SLOC for Duotyping systems are slightly more than traditional subtyping systems, the Duotyping systems come with extra features. Nevertheless the mechanization effort is roughly the same for version with and without Duotyping. Also, as illustrated in Sections 3 and 4, the vast majority of the lemmas/metatheory for calculi with Duotyping are similar to traditional systems with subtyping.

5.4 Does Duotyping make transitivity proofs simpler?

Transitivity is often the most difficult property to prove in the metatheory of a language with subtyping. Table 1 highlights a brief comparison between the techniques for the transitivity proof of various systems. Transitivity of systems with Duotyping is generally proved by induction on the Duotyping relation. One exception is $F_{F\diamond}$ where induction on the Duotyping does not work. As discussed in Section 2.5 Duotyping allows us to simplify the transitivity proof by using a different inductive argument.

Table 5 shows the SLOC for transitivity proofs of various systems. The SLOC for $\lambda_{<}$ transitivity proof are 7 and the SLOC for λ_{\diamond} transitivity proof are 4. Similarly, the SLOC for $F_{k<}^{\wedge\vee}$ transitivity proof are 38 and 18 for the transitivity proof of $F_{k\diamond}^{\wedge\vee}$. This evaluation shows that Duotyping always allows us to reduce the size of the transitivity proof. Again, it is important to note that Duotyping also provides extra features of lower bound and lower bounded quantification. Despite these additional features in Duotyping systems, their transitivity proofs are shorter than the traditional systems with subtyping.

■ **Table 5** SLOC for transitivity proofs.

Subtyping System	Transitivity SLOC	Duotyping System	Transitivity SLOC
$\lambda_{<}$	7	λ_{\diamond}	4
$\lambda_{<}^{\wedge\vee}$	13	$\lambda_{\diamond}^{\wedge\vee}$	11
$F_{k<}$	26	$F_{k\diamond}$	13
$F_{k<}^{\wedge\vee}$	38	$F_{k\diamond}^{\wedge\vee}$	18

However we could not employ this proof technique in our Duotyping version of full $F_{<}$ ($F_{F\diamond}$). The problem is related to *narrowing*, which in $F_{F\diamond}$ is closely coupled with *transitivity*. Despite that we could still apply the technique to most systems with Duotyping, and even for $F_{F\diamond}$ we can still prove transitivity using the same technique as in the traditional $F_{<}$ (i.e. using the middle type as the inductive argument).

5.5 Is Duotyping a generally applicable technique?

Our case studies indicate that Duotyping is generally an applicable technique. In all the systems that we have tried to use Duotyping, we have managed to successfully apply it. Furthermore we believe that Duotyping can be essentially applied to any system with a traditional subtyping relation. The most complex system where we have employed Duotyping is $F_{F\diamond}$. In $F_{F\diamond}$ universal quantification allows Duotyping between the bounds, generalizing the universal quantification presented in Section 4. Rule GDS-FORALLFFS in $F_{F\diamond}$ employs two operations $|\diamond_1|_{\diamond_2}$ and $A \tilde{\diamond} B$:

$$\begin{array}{l}
 |\diamond_1|_{\diamond_2} \\
 \quad |<|_{\diamond_2} = \bar{\diamond}_2 \\
 \quad |>|_{\diamond_2} = \diamond_2 \\
 A \tilde{\diamond} B \\
 \quad A \tilde{<} B = B \\
 \quad A \tilde{>} B = A
 \end{array}$$

$|\diamond_1|_{\diamond_2}$ takes two modes \diamond_1 and \diamond_2 as input, and flips \diamond_2 if \diamond_1 is subtyping, otherwise it returns \diamond_2 . This operation chooses the mode to check the relationship between the bounds of the two universal quantifiers being compared for Duotyping. The second operation $A \tilde{\diamond} B$ selects the bounds to use in the environment when checking the Duotyping of the bodies of the universal quantifiers. It takes a mode \diamond and two types $A B$ as inputs, and returns the second type if the mode is subtyping, otherwise it returns the first type.

6 Related Work

Apart from informally observing duality of type system features, as far as we known, formally exploiting duality in subtyping relations has not been investigated in the past. However there is plenty of work on uses of duality in programming language theory. Furthermore there is related work on type systems that exploit various generalizations for added expressive power or economy in metatheory and implementation. We discuss these next.

6.1 Duality in Logic and Programming Language Theory

In type theory [5] and/or category theory [30, 14] duality occurs in various forms. For instance, the duality between sum and product types is well-known in both type and category theory. Properties about such types often explicitly acknowledge duality. Many properties about sum types are presented as dual properties of corresponding properties on product types and vice-versa. Our Lemma 6 is an example of a property that applies to both union and/or intersection types. In this property duality is not only acknowledged, but directly exploited in the lemma itself to provide a generalized property that can be specialized to one construct and its dual. Various other dualities between constructs are known and exploited in various ways in type and/or category theory. For example, existential and universal quantification can be captured by an encoding by one through the other. The type $\exists\alpha. A$ can be encoded as $\forall\beta. (\forall\alpha. A \rightarrow \beta) \rightarrow \beta$, which requires a kind of CPS translation [20] of the corresponding terms. Similar encodings exist for sums and products.

In the field of *proof-theoretic semantics* [25] and in *natural deduction* the concept of *harmony* is used to describe introduction and elimination rules that are in some sense dual. For instance, the usual rules for introduction and elimination of conjunction are in perfect harmony. The inversion principles by Prawitz [38] are a general procedure to associate to any arbitrary collection of introduction rules a specific collection of elimination rules. The elimination rules are in harmony with the given collection of introduction rules. Prawitz inversion principles attempt to capture harmony in a more precise way, directly expressing it formally. Therefore inversion principles have similar considerations to **Duotyping** in terms of expressing some form of duality directly in a formalism. However inversion principles focus on introduction and/or elimination rules, while **Duotyping** is focused on subtyping. Nevertheless in future work we are interested in exploiting the use of duality in the typing relation more. We believe that the notion of harmony and inversion principles could be quite helpful in such work.

Double-line rules [21] are deduction rules that can be read both from top to bottom (as usual) and also from bottom to top. In other words they express two standard (dual) deduction rules in a single double-line rule. Like **Duotyping**, double-line rules aim at expressing a form of duality in a single rule. Unlike **Duotyping**, double-line rules are concerned with (dual) rules where the premises and conclusions of one rule become the conclusions and premises of the other rule, respectively.

Bernardi et al. [10] explain duality relations in the context in *session types*. Binary session types have two endpoints connected through one communication channel. In session types, connected endpoints should have a dual relation in their session types. The duality relation in session types is related to types and may have various interpretations. In contrast **Duotyping** is about subtyping (or supertyping).

The duality between data and codata is well-known in programming language theory [14]. Data types and codata types are duals in the sense that data types are defined in terms of constructors while codata types are defined in terms of destructors. More recently, such duality has been exploited in language design [35, 13] to provide an automatic way to switch between programs defined on datatypes and equivalent programs defined on codata types. The use of duality in this line of work is quite different from ours.

6.2 Generalizations in Type Systems and Type Theory

Pure type systems (PTSs) [45, 31, 2, 28, 41, 48] capture a generalization of various type systems ($F, F^\omega, \lambda P$). Typing rules of multiple type systems are expressed in pure type systems via parameterization. PTSs are parameterized by three sets: a set of sorts; a set of axioms;

and a set of rules. Concrete type systems (such as System F), are recovered with concrete instantiations of those sets. Pure type systems with subtyping [48] are a variant of pure type systems that captures a family of type systems with subtyping. This variant captures only the upper bounds. It does not provide subtyping generalization with both upper and the lower bounded quantification like our **Duotyping** generalizations of $F_{<}$. Pure subtype systems [26] is a family of calculi based on subtyping only (and without a typing relation). This system eliminates the need of typing and presents an alternative to typing using subtyping only. Pure subtype systems support upper bounded quantification, but no lower bounded quantification.

Modal Type Theory. Modal type theory [33] is an extension of type theory which provides type rules using modalities. Modal type theory can represent a proposition as types which may be proved based upon the deduction rules in a given context. Modal type theory also employs modes, for instance *possibility* and *necessity* [42, 33]. There are many type systems that use modes to generalize typing relations. One can view **Duotyping** as a simple instance of a relation with a mode. In **Duotyping** the mode is either subtyping or supertyping.

Bi-directional type checking. Bi-directional type checking [37, 23] also employs a mode, but in the typing relation instead. Bi-directional type checking is a common technique, used in implementations of programming languages, that can eliminate redundant type annotations. Bi-directional type-checking is also employed in several type systems, especially those where full type inference is undecidable [37, 24]. In such cases only partial inference methods are feasible in practice, which means that some type annotations are necessary. Bi-directional type checking is useful in such cases, allowing the type information to be easily propagated without requiring further (redundant) annotations. The modes in bi-directional type-checking are checking or synthesis. Checking checks a given term against a given type, whereas the synthesis infers the type based upon the available information in the context.

Unified Subtyping. Unified subtyping [46] is a technique that can be used in dependently typed systems supporting unified syntax to model typing and subtyping in a single relation. The single unified subtyping relation generalizes both typing and subtyping. Like **Duotyping**, unified subtyping can also help reducing language metatheory and duplication. However unified subtyping is orthogonal to **Duotyping** and does not exploit duality of features. We believe that both techniques can complement each other.

Bounded quantification and generalizations. System $F_{<}$ [16] is extensively studied due to its feature of bounded quantification. F-bounded quantification [15] is a generalization of bounded quantification to handle recursive types. Although we are not aware of an extension of $F_{<}$ with lower bounded quantification, such notion has appeared before in some calculi. For instance, Igarashi and Viroli [27] have pointed out correspondence between use-site variance and existential types and, in order to capture contravariance, they introduced lower-bounded existential types.

One generalization of $F_{<}$ is studied by Amin and Rompf [4], which formalizes *type bounds* in Scala. Type bounds is an interesting feature in Scala as elaborated by the following code (code extended from Section 2.4):

```
class TypeBoundsCollection[S >: GraduateStudent <: Student](obj: S) {
  def student: S = obj
}
```

While in our variants of $F_{<}$, we support either lower bounded quantification or upper bounded quantification (but not both at once), Scala’s type bounds allow both upper and lower bounds at once. This is clearly more expressive than what we have, but it comes with its own problems. Formalisms with Scala-like type bounds often need to include a transitivity axiom (and thus are non-algorithmic) and they have to deal with the bad bounds problem. In contrast our simpler extension of type bounds is comparable in complexity to $F_{<}$ ’s upper bounded quantification, and there is a set of algorithmic subtyping rules without a built-in transitivity axiom.

Intersection and Union Types. Intersection and union types [6, 22, 32, 36] are getting significant attention in recent years, and are used in several modern programming languages (including Scala, Flow or TypeScript). Reynolds [39] was the first to promote the use of intersection types in programming languages. Later on, Pierce [36] studied intersection types, union types and polymorphism combined in a typed λ -calculus. Recently, Muehlboeck and Tate [32] presented a generalized formulation of calculi with union and intersection types. They demonstrated it with the help of Ceylon programming language [29]. Dunfield [22] presented an expressive calculus with a merge operator and unrestricted intersection types with union types. We exploit the duality of union and intersection types to illustrate **Duotyping**. Our **Duotyping** calculi manages to capture the six common rules for unions and intersections using three rules only (plus the duality rule), which provides a simple illustrative example of the use of duality.

7 Conclusion

In this paper, we have presented a generalization of subtyping using a relation parameterized by a mode. We call this relation **Duotyping**. **Duotyping** allows formalizations of subtyping to exploit duality between features directly in the formalism, and provides multiple benefits over traditional subtyping relation. It shortens subtyping specifications, provides dual features essentially for free and simplifies the transitivity proof in many calculi. An example of an extra dual feature that is obtained for free, with a **Duotyping** design, is lower bounded quantification. Lower bounded quantification arises naturally when we apply the **Duotyping** to conventional $F_{<}$ type systems. To validate the benefits of **Duotyping**, we have conducted an empirical evaluation, implemented multiple calculi using both a traditional subtyping relation and a **Duotyping** formulation, and compared the resulting formalizations.

Duality has been studied extensively in several contexts in the past. However, as far as we know, our work is the first to study duality in the context of subtyping. Future work on **Duotyping** includes applying the **Duotyping** methodology to several other forms of subtyping that are of practical interest. We are particularly interested to apply **Duotyping** to systems that include a feature, but have no obvious dual feature. We hope to discover potentially new features that are useful for programming. Furthermore, we hope to scale the approach so that it can be used in real programming language implementations, leading to more compact and consistent implementations of subtyping. Another domain for future work on **Duotyping** is the typing relation. Although we already have some simple cases where **Duotyping** also provides benefits for typing (see Section 4), we acknowledge that introduction and elimination rules for some (dual) constructs introduce new challenges. For instance, more complex introduction and elimination forms for union types and intersection types can be quite different [22]. Further exploration is needed to see how much **Duotyping** can help in typing relations for calculi with such features. More generally, **Duotyping** promotes the use of

duality in language design. We envision that in the future new language designs can exploit duality to ensure consistency between various language constructs, by exploiting techniques similar to Duotyping.

References

- 1 Andreas Abel and Dulma Rodriguez. Syntactic metatheory of higher-order subtyping. In *International Workshop on Computer Science Logic*, pages 446–460. Springer, 2008.
- 2 Robin Adams. Pure type systems with judgemental equality. *Journal of Functional Programming*, 16(2):219–246, 2006.
- 3 Nada Amin, Adriaan Moors, and Martin Odersky. Dependent object types. In *19th International Workshop on Foundations of Object-Oriented Languages*, 2012.
- 4 Nada Amin and Tiark Rompf. Type soundness proofs with definitional interpreters. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*, POPL 2017, 2017.
- 5 Peter B. Andrews. *An Introduction to Mathematical Logic and Type Theory: To Truth through Proof*. Academic Press, Inc., 1986.
- 6 Franco Barbanera, Mariangiola Dezani-Ciancaglini, and Ugo Deliguoro. Intersection and union types: syntax and semantics. *Information and Computation*, 119(2):202–230, 1995.
- 7 Henk Barendregt, Mario Coppo, and Mariangiola Dezani-Ciancaglini. A filter lambda model and the completeness of type assignment 1. *The journal of symbolic logic*, 48(4):931–940, 1983.
- 8 Bruno Barras, Samuel Boutin, Cristina Cornes, Judicael Courant, Jean-Christophe Filliatre, Eduardo Gimenez, Hugo Herbelin, Gerard Huet, Cesar Munoz, Chetan Murthy, et al. The coq proof assistant reference manual: Version 6.1, 1997.
- 9 Jon Barwise and Robin Cooper. Generalized quantifiers and natural language. In *Philosophy, language, and artificial intelligence*, pages 241–301. Springer, 1981.
- 10 Giovanni Bernardi, Ornela Dardha, Simon J Gay, and Dimitrios Kouzapas. On duality relations for session types. In *International Symposium on Trustworthy Global Computing*, pages 51–66. Springer, 2014.
- 11 Jan Bessai, Boris Döder, Andrej Dudenhefner, Tzu-Chun Chen, and Ugo de’Liguoro. Typing classes and mixins with intersection types. *arXiv preprint*, 2015. [arXiv:1503.04911](https://arxiv.org/abs/1503.04911).
- 12 Gavin Bierman, Martín Abadi, and Mads Torgersen. Understanding typescript. In *European Conference on Object-Oriented Programming*, pages 257–281. Springer, 2014.
- 13 David Binder, Julian Jabs, Ingo Skupin, and Klaus Ostermann. Decomposition diversity with symmetric data and codata. *Proceedings of the ACM on Programming Languages*, 4(POPL):30, 2019.
- 14 Richard Bird and Oege de Moor. *The Algebra of Programming*. Prentice-Hall, 1996. URL: <http://www.cs.ox.ac.uk/publications/books/algebra/>.
- 15 Peter Canning, William Cook, Walter Hill, Walter Olthoff, and John C Mitchell. F-bounded polymorphism for object-oriented programming. In *FPCA*, volume 89, pages 273–280, 1989.
- 16 Luca Cardelli, Simone Martini, John C Mitchell, and Andre Scedrov. An extension of system f with subtyping. *Information and Computation*, 109(1-2):4–56, 1994.
- 17 Luca Cardelli and Peter Wegner. On understanding types, data abstraction, and polymorphism. *ACM Computing Surveys (CSUR)*, 17(4):471–523, 1985.
- 18 Arthur Charguéraud. The locally nameless representation. *Journal of Automated Reasoning*, 2011.
- 19 Avik Chaudhuri. Flow: a static type checker for javascript. *SPLASH-I In Systems, Programming, Languages and Applications: Software for Humanity*, 2015.
- 20 Oliver Danvy and Andrzej Filinski. Representing control: A study of the CPS transformation. *Mathematical Structures in Computer Science*, 2(4):361–391, December 1992.
- 21 Kosta Došen. Logical constants as punctuation marks. *Notre Dame J. Formal Logic*, 30(3):362–381, June 1989. doi:10.1305/ndjfl/1093635154.

- 22 Joshua Dunfield. Elaborating intersection and union types. *Journal of Functional Programming*, 24(2-3):133–165, 2014.
- 23 Joshua Dunfield and Neel Krishnaswami. Bidirectional typing. *arXiv preprint*, 2019. [arXiv:1908.05839](https://arxiv.org/abs/1908.05839).
- 24 Joshua Dunfield and Neelakantan R. Krishnaswami. Complete and easy bidirectional typechecking for higher-rank polymorphism. In *ICFP*, 2013.
- 25 Gerhard Gentzen. Untersuchungen über das logische Schliessen. *Mathematische Zeitschrift*, 39:176–210, 405–431, 1934.
- 26 DeLesley S. Hutchins. Pure subtype systems. In *Proceedings of the 37th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL 2010, 2010.
- 27 Atsushi Igarashi and Mirko Viroli. On variance-based subtyping for parametric types. In *ecoop*, pages 441–469, Malaga, Spain, June 2002. sv. To appear in *ACM Transactions on Programming Languages and Systems*.
- 28 LSV Jutting. Typing in pure type systems. *Information and Computation*, 105(1):30–41, 1993.
- 29 Gavin King. The ceylon language specification, version 1.0, 2013.
- 30 Saunders Mac Lane. *Categories for the Working Mathematician*. Number 5 in Graduate Texts in Mathematics. Springer, 2 edition, 1998.
- 31 James McKinna and Robert Pollack. Pure type systems formalized. In *International Conference on Typed Lambda Calculi and Applications*, pages 289–305. Springer, 1993.
- 32 Fabian Muehlboeck and Ross Tate. Empowering union and intersection types with integrated subtyping. *Proceedings of the ACM on Programming Languages*, 2(OOPSLA):112, 2018.
- 33 Aleksandar Nanevski, Frank Pfenning, and Brigitte Pientka. Contextual modal type theory. *ACM Transactions on Computational Logic (TOCL)*, 9(3):23, 2008.
- 34 Martin Odersky, Philippe Altherr, Vincent Cremet, Burak Emir, Sebastian Maneth, Stéphane Micheloud, Nikolay Mihaylov, Michel Schinz, Erik Stenman, and Matthias Zenger. An overview of the scala programming language, 2004.
- 35 Klaus Ostermann and Julian Jabs. Dualizing generalized algebraic data types by matrix transposition. In *European Symposium on Programming*, pages 60–85. Springer, 2018.
- 36 Benjamin C Pierce. Programming with intersection types, union types, and polymorphism, 2002.
- 37 Benjamin C. Pierce and David N. Turner. Local type inference. *ACM Trans. Program. Lang. Syst.*, 22(1), 2000.
- 38 Dag Prawitz. Proofs and the meaning and completeness of the logical constants. In *Essays on Mathematical and Philosophical Logic. Synthese Library (Studies in Epistemology, Logic, Methodology, and Philosophy of Science)*, 1979.
- 39 John C Reynolds. Preliminary design of the programming language forsythe, 1988.
- 40 Tiark Rompf and Nada Amin. Type soundness for dependent object types (dot). In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA 2016, 2016.
- 41 Paula Severi and Erik Poll. Pure type systems with definitions. In *International Symposium on Logical Foundations of Computer Science*, pages 316–328. Springer, 1994.
- 42 Alex K Simpson. The proof theory and semantics of intuitionistic modal logic, 1994.
- 43 Martin Steffen and Benjamin Pierce. Higher-order subtyping, 1994.
- 44 Tarmo Uustalu and Varmo Vene. Comonadic notions of computation. *Electronic Notes in Theoretical Computer Science*, 203(5):263–284, 2008.
- 45 LS van Benthem Jutting, James McKinna, and Robert Pollack. Checking algorithms for pure type systems. In *International Workshop on Types for Proofs and Programs*, pages 19–61. Springer, 1993.
- 46 Yanpeng Yang and Bruno C. d. S. Oliveira. Unifying typing and subtyping. *Proceedings of the ACM on Programming Languages*, 1(OOPSLA):47, 2017.

- 47 Francesco Zappa Nardelli, Julia Belyakova, Artem Pelenitsyn, Benjamin Chung, Jeff Bezanson, and Jan Vitek. Julia subtyping: a rational reconstruction. *Proceedings of the ACM on Programming Languages*, 2(OOPSLA):113, 2018.
- 48 Jan Zwanenburg. Pure type systems with subtyping. In *International Conference on Typed Lambda Calculi and Applications*, pages 381–396. Springer, 1999.

Safe, Flexible Aliasing with Deferred Borrows

Chris Fallin

Mozilla¹, Mountain View, CA, USA
cfallin@c1f.net

Abstract

In recent years, programming-language support for *static memory safety* has developed significantly. In particular, *borrowing and ownership* systems, such as the one pioneered by the Rust language, require the programmer to abide by certain aliasing restrictions but in return guarantee that *no unsafe aliasing can ever occur*. This allows parallel code to be written, or existing code to be parallelized, safely and easily, and the aliasing restrictions also statically prevent a whole class of bugs such as iterator invalidation. Borrowing is easy to reason about because it matches the intuitive ownership-passing conventions often used in systems languages.

Unfortunately, a borrowing-based system can sometimes be too restrictive. Because borrows enforce aliasing rules for their entire lifetimes, they cannot be used to implement some common patterns that pointers would allow. Programs often use pseudo-pointers, such as indices into an array of nodes or objects, instead, which can be error-prone: the program is still memory-safe by construction, but it is not *logically memory-safe*, because an object access may reach the wrong object.

In this work, we propose *deferred borrows*, which provide the type-safety benefits of borrows without the constraints on usage patterns that they otherwise impose. Deferred borrows work by encapsulating enough state at creation time to perform the *actual* borrow later, while statically guaranteeing that the eventual borrow will reach the same object it would have otherwise. The static guarantee is made with a *path-dependent type* tying the deferred borrow to the container (struct, vector, etc.) of the borrowed object. This combines the type-safety of borrowing with the flexibility of traditional pointers, while retaining logical memory-safety.

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

Keywords and phrases Rust, type systems, ownership types, borrowing

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2020.30

1 Introduction

Managing memory ownership properly is central to safe, correct programming in any programming language with a mutable heap. Parallel programs with shared memory can easily experience non-deterministic, undefined behavior if two concurrently-executing threads write to the same memory. Even sequential programs can experience subtle correctness issues related to memory ownership: for example, pointer invalidation occurs when a data structure traversal simultaneously mutates that data structure, leading to dangling or incorrect references.

Modern programming languages have developed support for managing ownership correctly by encoding various invariants statically in the type system. Many works propose to augment pointers with ownership information or capabilities (indicating temporary exclusive access) [5, 4, 3, 1, 6, 12, 15, 22, 20]. Another related approach categorizes heap objects into disjoint heap subregions, and annotates pointers to refer only to particular regions [8, 11, 3, 9]. Among languages in widespread use today, the Rust programming language [18] provides an *ownership and borrowing* system that adapts ideas from lexical regions [8] to annotate

¹ This work is independent of author's employer and author does not speak for Mozilla.



© Chris Fallin;

licensed under Creative Commons License CC-BY

34th European Conference on Object-Oriented Programming (ECOOP 2020).

Editors: Robert Hirschfeld and Tobias Pape; Article No. 30; pp. 30:1–30:26

Leibniz International Proceedings in Informatics



LIPICs Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

lifetimes on pointers. The language’s type system tracks “borrows” of an owned object – pointers taken to the object, or copies of those pointers – and ensures that only *one* mutable view of the object can be used at a time by blocking access to the original object for the borrow’s duration. All of these systems work to enforce memory safety in some way by either disallowing some pointers to *exist*, or else disallowing some pointers to be *accessed*, in order to prevent aliasing, concurrent accesses that would cause correctness issues.

Although these systems can improve correctness for many classes of programs, there is still resistance to their adoption because they are often not *flexible* enough. For example, since its release in 2015, the Rust language has become well-known for the introductory experience of “fighting the borrow checker”². The patterns that its ownership-passing and temporary-borrowing system allows and encourages are unfamiliar to programmers accustomed to unrestricted pointers in C or Java. As a result, Rust programmers have developed creative workarounds in their data-structure designs. For example, programs that manipulate graphs of objects, or otherwise have unpredictable heap graphs, sometimes use indices into a vector of objects as pseudo-pointers, in place of true pointers (borrows). Unfortunately, this merely sidesteps the problem, because these pseudo-pointers can be error-prone (as we will show later), and in any case are more cumbersome to use.

In this paper, we observe that an ownership and borrowing memory-management discipline is sometimes inflexible for *artificial and unnecessary reasons*, and that by splitting the borrow operation into two parts – the lookup, producing a “deferred borrow” handle, and a later conversion into a real borrow – much of the ease-of-use of an unrestricted language is recovered. The key idea in this work is to encapsulate a reference to an object that is not (yet) a borrow, so it does not trigger the restrictive mutual-exclusion rules that mandate only one usable mutable reference exist. Our language extension uses *path-dependent types* to statically bind this reference to the container (such as a vector, key-value map, struct or tuple) that owns the object. The deferred-borrow reference can later, at the point of actual use, be recombined with a reference to the container to get a true reference (borrow) of the pointed-to object. This true borrow lasts only as long as needed (during the use itself), and thus does not prevent the program from holding many deferred-borrow references to the same object, several of them mutable at once. Flexibility is achieved without giving up Rust’s memory safety or suitability for concurrency.

The structure of the remainder of this paper is as follows. In §2, we first provide a tour of the essential aspects of Rust’s borrowing and ownership system. Then, in §3, we show how Rust’s borrows do not provide as much flexibility as pointers do, and demonstrate the alternative approaches that programs often use. We show that while these approaches attain Rust’s memory safety properties in a *literal* sense, they lack what we call *logical memory safety*: the pseudo-pointers (e.g., indices into vectors), if used incorrectly, can cause the program to silently access the *wrong* data. We describe what would be necessary to avoid these logical memory safety violations. In §4, we introduce an extension to the Rust type system, *static path-dependent types*, that provide a minimal building-block for object references that retain logical memory safety. We prove that static path-dependent types provide an essential *object-binding* property. In §5, we finally introduce the concept of a *deferred borrow*, which is a general pattern that can be implemented by several types of *containers*, such as vectors, key-value maps, structs, and tuples. In §6, we describe how we have emulated static path-dependent types in the existing Rust type system for evaluation

² In the 2018 Rust community survey [19], the second and third most difficult-rated topics to learn were ownership/borrowing and borrow lifetimes, after only the macro system.

purposes. In §7, we analyze several common data-structure design patterns and qualitatively describe how the deferred borrows approach compares to other techniques. In §8 we consider related work.

2 Background: Ownership and Borrowing

In order to understand the need for deferred borrows, we first describe *borrowing-and-ownership-based memory management* as it is practiced in the Rust programming language [18]. We will describe how the language uses *ownership* to establish unique, unaliased access to heap objects, which permits statically-verifiable safe parallelism (§2.1). Then, because a pure ownership tree is cumbersome and difficult to use, we describe how Rust allows *borrowing* of owned objects by a form of lifetime-restricted pointer (§2.2). Finally, we look at how Rust container types typically use borrowing to encode safety invariants and extend the memory safety of core Rust data structures to the library level (§2.3).

2.1 Object Ownership and Linear Move Semantics

The Rust language, as many other memory safe languages or type systems before it [5, 4, 3], begins with establishing *unique ownership* for every object. An instance of an object (in Rust, a `struct`) can either exist on the stack or on the heap. In the former case, the local variable binding owns the object; in the latter, the heap memory is ultimately managed by an object on the stack, even if this is just a `Box<T>` (a pointer type).

The language uses *affine types* to ensure that a given object instance is not duplicated. When a non-copyable type is assigned (only primitive data types are trivially copyable), the object is *moved out of* the original storage slot (e.g., local variable binding), and the storage slot becomes inaccessible.

Fig. 1 shows a simple demonstration of moving ownership. We can see several examples of both stack-allocated and heap-allocated objects in this program. The `s` local variable is assigned an object initializer for a struct of type `S`; the storage for `s` is allocated in the stack frame. Several of its fields own heap storage, however: for example, the `t` field owns an instance of `T` that is stored on the heap, and the `children` field owns a vector (array-backed list) of `S`, also stored on the heap.

There are two important aspects to Rust’s ownership system on display here. First, without borrowing (which we introduce below), storage is organized into a strict *tree of access paths*. There is data at `s`, and `s.t`, and `s.children[0]`, and `s.children[0].t`. Each object has exactly *one* access path, however: there is no aliasing of heap references.

Second, Rust enforces *move semantics*. When the program assigns local variable `t` the value of `s.t`, the assignment *moves* the object, and `s.t` is no longer accessible. The program’s attempt to access `s.t.a` later is thus a static compilation error. Likewise, `s.children` becomes inaccessible after a move. These assignment semantics preserve the above property: by never duplicating non-trivial (non-copyable) data, we never create aliasing pointers. Of course, this restriction will be relaxed later, but only in a controlled way.

By providing exactly one path to any particular object, the ownership system can provide both (i) precise memory management – when a path goes out of scope, its subtree of memory resources is freed – and (ii) safe, deterministic (race-free) parallelism, by passing ownership of entire subtrees to separate parallel tasks. Despite these advantages, it is cumbersome: many common programming idioms rely on holding multiple references to particular heap objects. We now describe how pointers can be re-introduced in a limited but safe way.

30:4 Safe, Flexible Aliasing with Deferred Borrows

```
struct S {
    value: u64,
    t: Box<T>,
    children: Vec<S>,
}
struct T {
    a: u64,
    b: u64,
}

fn main() {
    // `s` lives on the stack, and is the unique owner of:
    // `t`, a `T` instance that is also on the stack
    // `children`, a `Vec` whose storage (and S instances)
    // are on the heap
    let mut s: S = S { value: 1,
                       t: Box::new(T { a: 1, b: 2 }),
                       children: vec![ /* ... */ ] };

    s.value += 1;
    s.t.a = 100;
    s.children.push(S { ... });

    let t = s.t;           // moves `t` out of `s`
    let children = s.children; // moves `children` out of `s`
    s.t.a = 1;           // ERROR! (s.t moved out of already)
}
```

■ **Figure 1** Rust ownership system: examples.

2.2 Safe Pointers: Lifetime-constrained Borrows

In order to allow references to a particular object, Rust permits *borrows*. A borrow is a kind of pointer that can be created from an owned access path, and that has an explicit *lifetime* that is statically restricted to maintain the safety properties that we introduced with unique ownership above. In particular:

- A borrow *temporarily restricts access* to an access path, just as a move out of that path permanently restricts access.
- A borrow can only exist as long as the original borrowed object exists: for example, it cannot be returned from a function if it borrows an object on that function's stack frame.

Both of these restrictions are implemented using *lifetimes*. A lifetime is, conceptually, a static description of a period of time during execution: either a local scope or a contiguous range of program points. Semantically, every borrow has a lifetime, and every local variable does, as well. In Rust's syntax, a lifetime is written as `'a`, and a borrow of an object of type `T` is written `&'a T`. Though every borrow semantically has a lifetime, lifetimes are usually inferred and so they can be omitted.

When a borrow is created, the borrow's lifetime is *constrained* such that the borrowed object must *outlive* the borrow, and the lifetime of the local variable that holds the borrow (as a value) must be outlived by the borrow lifetime. This is illustrated in Fig. 2.

Note that borrows can be included in data structures: when an aggregate datatype (such as a `struct`) has a member whose type is a borrow, the lifetime of that borrow must be defined as a *generic parameter* of the type. This need arises because such a type is (usually) defined outside of any function scope, and hence no lifetimes are otherwise in scope. This implicitly creates an outlives-constraint: when one object contains a pointer to (i.e., a borrow of) the other, the latter must outlive the former. This is analogous to the local-variable case above.

```

struct S { ... }

fn main() {
    let mut s: S = S { ... };

    s.a = 1;

    let mut s_borrow: &mut S = &mut s;           // \
                                                // |
    s.a = 2; // ERROR: s is currently borrowed // | borrow lifetime
                                                // |
    s_borrow.a = 2; // OK.                      // /

    // s_borrow is now dead -- `s` can be accessed again.

    s.a = 3;
}

```

■ **Figure 2** Rust borrowing examples.

2.2.1 Access Path-based Disjointness

The borrow system supports two types of borrows: *mutable* (as `&mut T`) and *immutable* (as `&T`). Multiple immutable borrows may have overlapping lifetimes, and read-only accesses to the original, borrowed, object may occur while such borrows exist. In contrast, if a mutable borrow is created, then no other mutable borrow of that object may have an overlapping lifetime, and the original object is inaccessible during its lifetime, as illustrated in Fig. 2.

In order to maintain this single-access-path invariant that enables safe parallelism, the Rust borrow system tracks *disjointness* of borrows. In particular, for any given object stored within a local variable or field of a struct (identified by an access path starting from a local variable), the compiler tracks which borrows are active for which contiguous spans of static program points, and ensures that no two incompatible borrows or direct accesses overlap.

2.2.2 Two Guarantees: Safety and Unique Mutability

The borrow system in Rust provides pointers that have been statically verified to retain two important properties: *memory safety* and *unique mutability*. Memory safety arises from outlives-constraints, and means that a pointer cannot be dangling; every pointer dereference thus accesses valid memory, and the program can never crash from an invalid pointer access. Unique mutability arises from disjointness constraints, and means that if a pointer is mutable (can be used for writes), then it is the only available access path to its pointee. This is what allows for safe parallelism: because of unique mutability, a program cannot have data races. Both sorts of constraints, and both resulting guarantees, also allow for the creation of safe *container types* that extend the access path-based system to dynamically-sized, heap-resident containers.

2.3 Borrows and Container Types

The power of Rust’s ownership and borrow system arises from the way in which its basic primitives – borrows with lifetimes, and constraints between lifetimes – can be used by *libraries* to build type-safe containers.

Consider, for example, the simple vector (array-backed ordered list with $O(1)$ element access) API in Fig. 3. This API has two functions: `get_index_mut`, which returns a pointer to a storage slot within the vector, and `append`, which appends a new element.

30:6 Safe, Flexible Aliasing with Deferred Borrows

```
struct Vector<T> { ... }
impl<T> Vector<T> {
    // The 'a lifetime (usually implied, but written explicitly here) ties
    // the returned borrow to an implicit borrow of `self` at the callsite.
    // The callee cannot use the `Vector` in any other way while the
    // borrow to the element is live.
    fn get_index_mut<'a>(&'a mut self, index: usize) -> &'a mut T { ... }
    fn append(&mut self, t: T) { ... }
}

fn main() {
    let mut v = Vector<u32>::new(...);

    let elem0 = v.get_index_mut(0); // borrows `v`, returns borrow to elem
    *elem0 = 1;
    // `elem0` borrow now dead (not used below). Borrow lifetime on `v` ends.

    let elem1 = v.get_index_mut(1);
    let elem2 = v.get_index_mut(2); // ERROR: `v` already borrowed mutably.
    *elem1 = 2;
    *elem2 = 3;

    let elem3 = v.get_index_mut(3);
    // The mutable borrow also serves to "freeze" the underlying storage
    // location in place (no other method on the `Vector` can be invoked,
    // because we cannot borrow its `self` again). This is needed
    // because the borrow is just a pointer: the container must not
    // e.g. reallocate its storage to grow an array, rehash a table or
    // rotate a tree, etc.
    v.append(4); // ERROR: `v` already borrowed mutably.
    *elem3 = 4;
}
```

■ **Figure 3** Rust lifetime constraints used in container APIs.

In order to return a borrow to internal storage – an element contained within, and whose memory is managed by, the vector – the `get_index_mut` function must take a borrow of the vector itself. Given the borrow of the *whole* vector, it can safely return a borrow of a piece of that vector, as long as the returned borrow’s lifetime is contained within the original borrow’s lifetime (trivially true here as the lifetime is the same `'a`).

This lifetime-outlives constraint, relating a borrow of the original container to that of its element, serves two important purposes. First, the borrow on the container serves as a proxy for the borrow of the element itself. Because the Rust borrow-checker does not have a precise understanding of vector indices or hashmap keys (for example), it cannot directly track the access-paths that name these storage locations. Hence, although it might be perfectly valid (assuming a well-behaved data structure implementation) to borrow both `v[i]` and `v[j]` mutably if $i \neq j$, because the element storage slots are disjoint, the borrow checker cannot actually verify this. The container API leverages the borrow checker in a sound but conservative way, requiring a borrow of the *entire container* when an element is borrowed. Because it is always sound to “over-borrow,” this maintains the no-dangling-pointer and no-aliasing-mutable-pointer safety properties of Rust’s type system.

Second, and just as importantly, this lifetime constraint and container-borrow mechanism *freezes the container layout in place* so that the raw pointer (which is how a borrow is implemented) remains valid. Here, any access with `get_index_mut` borrows the container mutably, which prevents any other access to the original object. Idiomatic container APIs also allow immutable element borrows, which borrow the container immutably: this allows other

read-only access to the container, but still prevents any mutation. This works because any other function that mutates the container – e.g., `append`, which might cause the underlying storage to be reallocated if more space is needed – takes a `&mut self` parameter, requiring a mutable borrow, which is incompatible with the outstanding (immutable or mutable) borrow.

3 Inflexible Borrows and Workarounds

Now that we have seen how Rust enables safe aliasing through *borrows* with carefully checked disjoint-lifetime and disjoint-mutability properties, let us consider how these limitations impact program design.

3.1 Borrowing-Incompatible Data Structures

Two basic patterns of pointer-based data structure design are problematic when borrows are used in place of pointers, corresponding to each of the constraints that the borrowing system imposes.

First, the borrowing system requires the borrowed object to *outlive* the borrow itself, to preserve the language’s memory-safety property. When one object points to another, the lifetime of the former must be strictly shorter than the latter. This immediately rules out *cycles* of borrows. Common data structures that are cyclic, such as graphs, doubly-linked lists, and trees with parent pointers, thus cannot be implemented in safe Rust.

Second, the borrowing system requires borrows to be *safe* relative to each other, and in particular, allows no more than one mutable borrow to exist at a time. There are many programming idioms that require holding pointers to inner elements of a data structure: for example, a “secondary index” might refer to elements in a vector or map indexed by an alternate key, or an algorithm may keep a stack of pointers to nodes as it traverses a graph or tree. Some of these pointers may later be used to update the data structure. Unfortunately, Rust cannot allow these pointers (borrows) to be mutable.

Both of these problems can be seen in Fig. 4. Program 1 shows a simple graph-manipulation program in C++, demonstrating the ease by which the graph node type can be defined. In contrast, in Program 2, we run into trouble as soon as we try to define the node types, caused by both reference cycles and aliasing mutable borrows. We clearly cannot carry over our habits of freely handling pointers as we had done in C++.

One approach, sometimes seen in core data-structure libraries, is to use “unsafe” raw pointers that circumvent the type system. While the Rust language allows this C-like flexibility via an escape hatch, the memory safety then relies completely upon the programmer’s care. We thus do not consider this approach further.

3.2 A Solution: Pseudo-Pointers

A common approach to allow arbitrary object references in safe Rust is to use names for objects that are not actually borrows (pointers), such as indices in a vector. Program 3 in Fig. 4 demonstrates this approach. Node references are by indices into a vector, and this vector is the true owner of the nodes. Rust’s tree-ownership model that described in §2 is retained, and the program is completely memory-safe. There is no issue when `visitEdge` needs to take references to two different nodes to mutate, because these references (indices) are not actually potentially-aliasing borrows, only integers. We call these integers *pseudo-pointers*.

Program 1: C++, using unrestricted native pointers

```
// Define a graph as a list of pointers to nodes; define a node's edges
// simply as pointers to other nodes.
typedef vector<Node*> Graph;
struct Node { vector<Node*> outEdges; };
void visitEdge(Node* n, Node *neigh) { ... }

void updateGraph(Graph& g) {
    for (Node* n : g)
        for (Node* neighbor : n->outEdges)
            visitEdge(n, neighbor);
}
```

Program 2: Rust, using references (borrows)

```
// Problem 1: we will not be able to construct a graph instance of Node<'a>
// because each node needs to outlive its pointed-to nodes; the cyclic
// dependency is impossible to resolve.
//
// Problem 2: we cannot hold mutable borrows of neighboring nodes in
// `outEdges`, because more than one borrow might exist (the type
// system will not allow us to create these borrows).
type Graph<'a> = Vec<Node<'a>>;
struct Node<'a> { outEdges: Vec<&'a mut Node> }
```

Program 3: Rust, using node indices

```
// Define a graph as an owned vector of nodes; define out-edges as
// indices of other nodes in this vector.
type Graph = Vec<Node>;
type NodeIndex = usize;
struct Node { outEdges: Vec<NodeIndex> }

// NOTE: this is cumbersome: every access to a node `n` is really
// `g.nodes[n]`.
fn visitEdge(g: &mut Graph, n: NodeIndex, neighbor: NodeIndex) { ... }

fn updateGraph<'a>(g: &mut Graph<'a>) {
    for n_idx in 0..g.len() {
        // Note: we need to copy the outEdges list here because `visitEdge`
        // below takes temporary mutable ownership of the entire graph `g`.
        let neighs = g[n_idx].outEdges.clone();
        for neigh_idx in &neighs {
            visitEdge(g, n_idx, neigh_idx);
        }
    }
}
```

■ **Figure 4** A graph-processing program, in C++ (first program) and Rust (second and third programs), that demonstrates the difficulties imposed by borrowing (second program) and the type-unsafety of the usual workaround (third program).

However, this approach has several downsides. First, and most directly, it is *cumbersome*. To make it work, we need to (i) pass a borrow to the true owner (here, the `Graph` object) everywhere along with the pseudo-pointers, and (ii) explicitly dereference the node by indexing the vector at each point of use. However, beyond the mere ergonomics issues, a potential *correctness* issue looms: the vector access `g[n_idx]` may not refer to the same `Node` at access time that it did when the index was taken! If, for example, the program removes a node and compacts the node-vector, all node indices become invalid, but the type system does not prevent their use. Even worse, if the program contains *multiple* vectors of the appropriate type (say, the program maintains several graphs simultaneously), an index intended for one may be used to access another. We define a new term to encapsulate these issues: *logical memory safety*.

3.3 Logical Memory Safety

Rust provides *memory safety* in the sense that any memory accessed by a Rust program must be valid memory and must be a valid, still-live instance of the object implied by the type of the pointer (borrow). Nothing in the vector-based approach invalidates this property, nor could it, because the guarantee is true for any safe Rust program. Unfortunately, nothing in the Rust type system ensures that the *correct* memory will be accessed, because correctness is a program-specific property.

We define *logical memory safety* in the context of a program that uses pseudo-pointers to mean that every access to an object via a reference (such as a vector index) accesses the same object that the reference was created to refer to. This property provides essentially the same guarantee that a borrow does: a borrow also ensures that the pointed-to object remains accessible, and remains *the same object*, during the lifetime of the borrow.

3.4 Maintaining Safety: Deferred Borrows with Irrevocable Binding

We can now concisely state the goal of this work: we wish to provide logical memory safety for object references without the limitations of borrows, i.e., in a way that retains the flexibility of pseudo-pointers.

We build on pseudo-pointers, because they resolve the conflicting-borrows problem right away. This is because they *defer* the actual borrow of the container until the referred-to object is used. In other words, `nodes[node_idx]` borrows `nodes` mutably, but only for as long as the particular operation on this node. This property is the origin of our term *deferred borrow*, which we expand further in §5.

To make pseudo-pointers logically memory-safe, let us consider what would be needed: in concrete terms, for the vector-based approach, we must ensure that an object's index in the vector is constant once added (by only appending to the vector). In addition, we must ensure that an index created for some particular vector is only ever used to access that vector, and not another, even if their static types match.

We can provide the append-only property at the library API level by encoding the invariant into the types: for example, provide a `.to_append_only()` method on `Vec<T>` that consumes the `Vec` (as we can do with linear types!) and returns an `AppendOnlyVec<T>`.

Ensuring that indices are only used with a *particular* vector, however, is more challenging. The existing Rust type system cannot encode this restriction. We must somehow *irrevocably bind* an index with a vector object, and require this binding when the access `node[node_idx]` occurs. In the following section, we now show how this can be done with *static path-dependent types*.

4 Static Path-Dependent Types in Rust

Our key contribution to the core Rust language that enables deferred borrows is the *static path-dependent type*. Path-dependent types have been proposed previously in a dynamic context [7, 14, 2], e.g. in Scala: in that context, a type is an element of some class, and each class instance has a *different* type (i.e., the dynamic object identity is part of the type). In contrast, our path-dependent types are static. This is an extension to an ordinary type that ties a value to *another particular value in-scope*, by its access path (local variable plus struct field(s)).

We define a type `T/x` to be a subtype of `T` that has the *path* `x`. Intuitively, a path can refer to any *storage place* that the borrow-checker tracks: e.g., a local variable binding or a subfield of one. The path can then be used to constrain function arguments so that, e.g., a

value of type T/x can only be combined with exactly the value in local x . This provides the necessary conditions for logical memory safety of deferred-borrow smart pointer objects, as each can be paired with the container from which it must eventually borrow.

4.1 Types with Static Dependent Paths

We augment the type system of Rust so that any type τ can be annotated with a path p to form type τ/p . The path describes a *storage slot*, or *place* in the terminology of Weiss et al. [21]’s formulation. Concretely, this is a *root binding* optionally extended with a path of struct fields. A root binding is (i) a local immutable variable binding in scope, within a function body; (ii) a formal parameter index, within a function type; or (iii) the *self* root, within a struct. In each of these contexts, the path corresponds to a fixed location that will hold the same value until it goes out of scope or is moved out of.

To communicate our intended semantics, we sketch a set of definitions and inference rules in Fig. 5. This scheme is built on top of that of Weiss et al. [21] for the *Oxide* language, which describes a small core language that captures Rust’s ownership and borrowing semantics. We first extend the type unification and subtyping judgments with rules to allow the path annotations on types to flow through the program. We extend the expression typing judgment to weaken path-dependent types when the corresponding path is dropped or moved out of, or when a mutable binding is modified: formally, when the place π is removed from the outgoing typing context Γ in T-MOVE’, we weaken any type τ/π in the typing context to simply τ . Finally, we modify the typing rules for function application and struct-field projection to translate the *roots* between the three domains (locals, function parameters, or a struct’s *self*). This allows struct field types and function parameter types to naturally refer to “neighboring” values, as we will see below.

4.2 Static Path-Dependent Types in Rust: Syntax and Examples

We now show how static path-dependent types might appear in several Rust snippets to give a flavor of their integration into the language. First, consider that we have a struct definition:

```
struct S<'a> { c: &'a Container, r: Option<ContainerRef/self.c> }
```

This struct holds a borrow (over whose lifetime it is parameterized) to some `Container` type that we have presumably defined elsewhere. It also holds a value of type `ContainerRef`. In this struct definition, however, we have augmented this type with a path `self.c`. In a struct field type context, any path on a path-dependent type must start with the `self` path prefix, and this prefix indicates that the following path refers to a neighboring field in the same struct instance. Here, whatever `ContainerRef` that is stored in `r` of a given instance will be irrevocably tied via its path to the container in `c`.

To use this value, we might write a function like the following:

```
fn foo() {
  let c = Container::new_with_contents();
  let mut s = S { c, r: None };
  let r = s.c.deferred_index(/* index = */ 42);
  s.r = Some(r);

  let elem = s.c.defborrow(s.r.unwrap());
  // ...
}
```


Static Path-Dependent Types

Note: This formalization extends that of Oxide [21], which captures the Rust ownership and borrowing system in a small core language. We omit a repetition of most of Oxide's rules and definitions, and show only those relevant to tagged types below.

e	Expression	x	Variable/Identifier	p	Path
τ	Type	π	Storage Place	i, n	Naturals

Language Extensions

$\tau ::= \dots \mid \tau/p$	Path-dependent type
$p ::= \pi$	Storage-place path (in expression context)
$\mid i.x_1 \dots x_n$	Formal parameter path (in function type)
$\mid \text{self}.x_1 \dots x_n$	Struct path (in struct field type)

$$\boxed{T; \Sigma; \Delta; \Gamma \vdash \pi : \tau \Rightarrow \Gamma'} \quad (\text{Typing judgment})$$

$$\text{T-MOVE}' \frac{\Gamma \vdash_{\text{uniq}} \pi : \tau^s \quad \tau_s \text{ noncopyable}}{\Sigma; \Delta; \Gamma \vdash \pi : \tau^s \Rightarrow \text{filter}(\Gamma - \pi, \pi)}$$

Similar adaptation to T-LET, omitted for brevity: filter path x from Γ in the out-context of let $x \dots$

$$\text{T-APP}' \frac{\Sigma; \Delta; \Gamma \vdash e_f : (\tau_1^s, \dots, \tau_n^s) \rightarrow \tau_f^s \Rightarrow \Gamma' \quad \Sigma; \Delta; \Gamma_{i-1} \vdash e_i : \text{funcpath}(\tau_i^s, (e_1, \dots, e_n)) \Rightarrow \Gamma_i, 1 \leq i \leq n}{\Sigma; \Delta; \Gamma \vdash e_f(e_1, \dots, e_n) : \tau_f^s \Rightarrow \Gamma_n}$$

places- typ meta-function modified so that struct-field types are filtered through structpath; if $x : \{x_1 : \tau/\text{self}.\pi, \dots\} \in \Gamma$, then $x.x_1 : x.\pi \in \text{Gamma}$.

$$\begin{aligned} \text{filter}(\Gamma, \pi) &= \Gamma[\tau/\pi \dots \mapsto \cdot] \\ \text{funcpath}(\tau, (e_1, \dots)) &= \tau[\tau'/i.\pi \mapsto x.\pi \text{ if } e_i = x] \\ \text{structpath}(\tau, \pi) &= \tau[\tau'/\text{self}.\pi' \mapsto \pi.\pi'] \end{aligned}$$

$$\boxed{\tau_1 \sim \tau_2 \Rightarrow \tau} \quad (\text{Type unification})$$

$$\text{U-PATH} \frac{\tau_1 \sim \tau_2 \Rightarrow \tau}{\tau_1/p \sim \tau_2/p \Rightarrow \tau/p}$$

$$\boxed{\Sigma \vdash \tau_1 <: \tau_2 \rightsquigarrow \delta} \quad (\text{Subtyping})$$

$$\text{S-PATH} \frac{\Sigma \vdash \tau_1 <: \tau_2 \rightsquigarrow \delta}{\Sigma \vdash \tau_1/p <: \tau_2/p \rightsquigarrow \delta}$$

■ **Figure 5** Static path-dependent types: definitions and rules as an extension of the Oxide formalization [21].

30:12 Safe, Flexible Aliasing with Deferred Borrows

In this function, we create an instance of `S`, and initially fill its field `c` with a new `Container`. Let us say that the method `Container::deferred_index` returns a value of type `ContainerRef/s.c` (we will see how such a function can be declared below). Then `r` has this type as well. This type unifies with the type of field `s.r` because `T/self.c` in struct-field type context maps to `T/s.c` for the struct at the particular place `s`.

How do we pass these path-qualified values between functions? Analogously to our approach for struct-field types, we allow *parameter* roots in parameter (and return value) types. Thus the type of one parameter can be bound to the value of another parameter. For example, we might define `deferred_borrow` above as follows:

```
impl Container {
  fn deferred_index(&self, index: usize) -> ContainerRef/0 {
    ContainerRef { /* ... */ }
  }
}
```

The return type `ContainerRef/0` refers to the 0-th parameter, in this case `self`: thus, the returned `ContainerRef` is bound to the passed-in `Container`.

Paths can always be stripped from types, but cannot be added in ordinary value dataflow, due to the subtyping rule `S-PATH` which specifies that $\tau/p <: \tau$. A value acquires a path at construction time: e.g., above, the return value is constructed with the struct-literal form for `ContainerRef`, and implicitly has type `ContainerRef/0`.

Finally, the function `defborrow` can place a path on an inbound parameter type, requiring that parameter to have the path in order for the function call to typecheck. In this running example, `defborrow()` requires the passed-in `ContainerRef` to be associated with the `self` `Container` in order for the call to succeed:

```
impl Container {
  fn defborrow(&self, r: ContainerRef/0) -> &Elem {
    // ...
  }
}
```

In summary, we see that static path-dependent types qualify arbitrary types with *storage paths*. The paths available depend on the context. For an expression in a function body, these are precisely the local storage paths, rooted at local variables. For a struct definition's fields, these are paths rooted at `self` and a field name in the same struct. Finally, for function parameter and return types, these are the numbered roots 0 to `n-1` referring to the `n` parameters of the function.

4.3 Correctness: Value-Correspondence Lemma

Building on the above intuition for “binding” one value to another, we can now describe the condition that path-dependent types ensure:

► **Lemma 1.** *If a storage place π is in scope with any type and with value v at a program point defined by typing environment Γ , and another storage place π' is in scope with type τ/π and with value v' at the same program point, then for any dynamic execution, the value v' is stored in a place of type $\tau' <: \tau/\pi$ only as long as the value v remains in storage place π .*

Proof sketch. Follows from the dynamic semantics. As long as the binding π it remains in scope, the value remains v (because bindings in Oxide are immutable). The rules `T-LET` and `T-MOVE` in the Oxide formalization [21] on which our formalization is built ensure that π is dropped from the typing context Γ when it is moved out of or when it falls out of scope. Our modifications to these rules apply our meta-function `filter()` to the typing context, weakening any type τ/π to simply τ . ◀

4.4 Generics and Path Parameters

For brevity, we have not included a formalization of *generic path parameters*, though we describe them informally here. In order to allow data structures to contain path-dependent types that refer to paths *outside* of the `struct` in question (i.e., outside of the `self` path), we allow generic parameters to provide paths. Field types are known at instantiation time according to the actual path provided as a parameter, as for type and lifetime generics.

4.5 Alias Analysis and Type-Tag Propagation

We note briefly that this type system can be seen as the combination of path-dependent types (in the dynamic sense of earlier work [7, 14, 2]) with a static must-alias analysis, so that all dynamic checks are replaced with static reasoning at type-check time. The rules for unifying dependent paths attached to types essentially form a very simple path-based intraprocedural must-alias analysis. When seen this way, one can also imagine several precision enhancements by adopting more advanced static must-alias techniques, as in e.g. Kastrinis et al. [10].

5 Deferred Borrows

In §3, we saw how the existing Rust borrow system can be inflexible in the face of some common program design patterns. We described the foundation of a solution in §3.4, describing how one might combine an index-based scheme (what we call *pseudo-pointers*) with some sort of strong *binding* between these indices and the particular container instances to which they refer in order to attain *logical memory safety*, a stronger property than Rust's language-level memory safety. Now that we have introduced static path-dependent types, we can show how to use these to achieve exactly this goal.

Our *key insight* in this work is that we can alleviate the inflexibility of the borrow system, caused by the conflict of multiple outstanding borrows, by avoiding a borrow until the point of use while retaining safety in other ways. This is the origin of the name *deferred borrow*.

A deferred borrow is an *API concept* that uses path-dependent types to provide a more flexible interface to a container. As we described in §2.3, borrows of container elements perform proxy borrows (borrows with the same lifetime) of the *entire* container; this approximation is necessary to retain memory safety because the borrow checker cannot reason about abstract index spaces or storage locations such as vector indices. The idea of a deferred borrow is to:

1. Freeze the *existence* of the reference element (e.g., disallow element deletion); and
2. Return some state that can allow a lookup and true borrow of the element later, even if the internal storage of the container has been rearranged in the meantime. This later lookup performs a borrow of the entire container, as an ordinary element reference does.
3. Tie this state to the container with a path-dependent type.

This strategy provides all the same guarantees as a true Rust borrow. First, while the true borrow is outstanding at the point of use, we have memory safety simply by reduction to the usual Rust container access idiom: the entire container is borrowed for the duration of the element access. However, for the *entire existence of the deferred-borrow element reference*, we also have a substantially similar guarantee: (i) the container is put into a state so that the element cannot disappear; (ii) the element reference is tied to this particular container; so (iii) when the deferred borrow is converted into a real borrow, the borrow will refer to exactly the desired element.

It is important to note that all of these properties were provided by a *true* borrow simply because a true borrow performs a borrow of the entire container for the duration of the element access. In contrast, a deferred borrow *synthesizes* this same guarantee from pieces. By doing so, without holding an outstanding borrow on the container, many of the unnecessary restrictions are avoided. In particular, while a deferred borrow exists, any of the container elements can be accessed or mutated, even if the outstanding deferred borrow also allows mutable access; and, if the container is implemented properly, the program can also append new elements to the container. In other words, we separate a “bookmark” phase, in which an element is identified, from a “use” phase, in which it is exposed for access.

5.1 Definition and Correctness Conditions

A deferred borrow idiom properly implemented by a container grants *logical memory safety*, as long as the container upholds the contract: a deferred borrow object returned by a lookup operation *must* convert into the same borrow at any future point if dereferenced with the same container object. Combining the value-correspondence lemma of §4.3 with this contract, we have the full guarantee to the user of the container API.

In slightly more precise terms, we can define a deferred-borrow implementation as an API pattern with the following conditions. Given a mutable container datatype that contains values of type V indexed by keys of type K and provides the following operations:

- `insert(c, k, v)`, which inserts a new *storage slot* dynamically into container c at abstract address, or key, k with initial value v ,
- `remove(c, k)`, which removes a key k from c ,
- `immutable_borrow(c, k)` which returns an immutable borrow (pointer) to the storage slot for k , and has lifetime constraints such that c is immutably borrowed as long as the returned borrow is in scope,
- `mutable_borrow(c, k)` which likewise returns a mutable borrow tied to a mutable borrow of the container,

and the usual key-value map semantics (the value seen under the pointer returned by `immutable_borrow` or `mutable_borrow` is that value last stored to a pointer fetched by `mutable_borrow` for that key), a deferred-borrow pattern is implemented with the operations:

- `deferred_get(c, k)`, which returns some abstract reference type r , tied to the container with a path-dependent type,
- `deferred_borrow(c, r)`, which given any r returned by `deferred_get(c, k)` at *any* point in the past with no interceding `remove(c, k)` operation, returns an immutable borrow to the storage slot *currently* backing k , with lifetime tied to a full-container borrow as above, and
- `deferred_get_mut(c, k)`, likewise but with mutable borrows.

The most important aspect of a deferred-borrow implementation is the property that a reference *remains valid* even if the container is later mutated. This implies that the abstract reference type must somehow keep a *logical* notion of storage-slot address, rather than a true pointer, unless the implementation can guarantee that the storage layout will never change. It is exactly this property that allows us to retain logical memory safety without freezing the container completely with an ordinary borrow.

The correctness of the deferred-borrow concept arises largely by definition from the above, in concert with the value-correspondence lemma of §4.3. By using path-dependent types on the interface above, a deferred-borrow implementation can statically ensure that a reference r produced from a particular container c is only ever used with that container. The application in concrete type terms is simple, and will be shown in the next section.

■ **Table 1** Examples of containers and associated element-reference (deferred borrow) types, with implementation strategies. A library that provides deferred-borrow types may preserve memory safety in several ways, trading off allowed mutation with the amount of state that is kept in the reference and the amount of (deferred) work to convert it into a true borrow.

<i>Base Container</i>	<i>Derived Type</i>	<i>Deferred-Borrow Element Reference</i>		
		<i>Reference State</i>	<i>Deferred Work</i>	<i>Borrow Result</i>
Vec<T>	Vec	index	bounds check; base plus index	Option<&T>
	AppendOnlyVec	index	base plus index	&T
	FrozenVec	true pointer	none	&T
HashMap<K, V>	HashMap	key, lookup hint	hash-table lookup	Option<&T>
	AppendOnlyHashMap	key, lookup hint	hash-table lookup	&T
	FrozenHashMap	true pointer	none	&T

5.2 Containers and Deferred-Borrow APIs

Let us now see how deferred borrows can be implemented concretely. First, we define a Rust trait that generalizes over any state that, in association with some container, can be converted into a borrow of an element in that container:

```
trait DefBorrow<T, Container> {
  fn def_borrow<'a>(&self/1, cont: &'a Container) -> &'a T;
}

trait DefBorrowMut<T, Container> {
  fn def_borrow_mut<'a>(&self/1, cont: &'a mut Container) -> &'a mut T;
}
```

These definitions simply mean that a particular “deferred borrow” object is associated with a *particular* container, and if a method on the deferred-borrow state is invoked with that container (invoking it with any other container instance is a type error), it will return a reference to (borrow of) the element. This borrow creates a true borrow of the container, but only as long as this particular access needs it; e.g., a program may hold many mutable deferred borrows, some of them aliasing, and dereference each in turn to perform a single mutation before dropping the true borrow.

How might this interface be implemented? Let us consider several real container types: the `Vec` (vector) and `HashMap` (key-value map built with a hashtable). Table 1 summarizes several options for each.

Recall that we need to *freeze the existence of the referred-to element* when the deferred borrow is created. A container whose element index space can dynamically shrink and grow (true for both `Vec` and `HashMap`) might do so in one of several ways. It could freeze its structure entirely, not allowing addition or removal, or it could still allow addition, simply prohibiting element removal.

In the first case, if any insertion or removal is prohibited, any actual pointer that refers to the internal storage for a particular element should remain valid, because the container will not need to reallocate to grow, and so the deferred-borrow object can carry an actual pointer. The abstraction is thus erased at runtime, and serves only to translate accesses to one of a large collection of true pointers into borrows of the container to ensure mutual exclusion.

30:16 Safe, Flexible Aliasing with Deferred Borrows

```
impl Vec<T> {
    // ...
    pub fn to_append_only(self) -> AppendOnlyVec<T> {
        AppendOnlyVec { vec: self }
    }
}

pub struct AppendOnlyVec<T> {
    vec: Vec<T>,
}

impl AppendOnlyVec<T> {
    pub fn deferred(&self, index: usize) -> AppendOnlyVecRef<T>/0 {
        AppendOnlyVecRef { index }
    }

    pub fn push(&mut self, t: T) {
        self.vec.push(t);
    }
}

pub struct AppendOnlyVecRef<T> {
    index: usize,
    _phantom: PhantomData<T>, // keep the Rust type-checker happy by using T.
}

impl DefBorrow<T, AppendOnlyVec<T>> for AppendOnlyVecRef<T> {
    fn def_borrow<'a>(&self/1, cont: &'a AppendOnlyVec<T>) -> &'a T {
        &cont.vec[self.index]
    }
}
```

■ **Figure 6** Excerpt of the implementation for `AppendOnlyVec`, one variant of a vector that implements deferred borrows. This variant ensures the existence of elements that have outstanding deferred borrows simply by disallowing element removals. Deferred borrows are just vector indices internally; type erasure makes this approach equivalent to “pseudo-pointers” at runtime, but it is more type-safe.

In the second case, however, the container is still allowed to grow by insertion, and so it must compute the element location only when the deferred borrow is converted into a true borrow. In this case, the deferred borrow will contain the *logical* element address – e.g., a vector index. This essentially emulates the pseudo-pointer idiom, but with more type-level safety.

Finally, if we extend the notion of a deferred borrow to one that can return an *optional* borrow (i.e., an element borrow or `None`), we can allow even a standard container with insertions and removals to produce deferred borrows. In this case, we also must retain a logical address only in the deferred-borrow object, and perform the lookup late.

Referring again to Table 1, these container variants can be seen as a form of typestate encoding the restrictions on container mutations that are allowed. The library user can convert containers only to more constrained variants. Each base type has `.to_append_only()` and `.to_frozen()` methods that consume the original (i.e., have non-borrowed `self` arguments) and return the appropriate constrained type, and the append-only type has `.to_frozen()` as well.

To provide a complete example, we show a simple implementation of `AppendOnlyVec`, the variant of the vector that allows insertions but not removals, in Fig. 6. The main highlights are that deferred-borrow objects reduce simply to vector indices at runtime (there need not

be any dynamic checks that the “correct” vector is accessed, because the path-dependent types ensure that statically), and that the vector indexing operation at true-borrow time can be assured of success because the underlying vector is not allowed to shrink after any deferred-borrow objects are produced.

Note that these types are not exhaustive by any means: one can imagine several other variants that make still different tradeoffs. For example, a container might allow individual element deletions yet still provide a deferred-borrow type that returns a `&T` rather than `Option<&T>` by dynamically tracking which elements have outstanding element references. Internal data structure design may also facilitate the creation of more efficient element references with less deferred lookup work: for example, a container might allocate stable memory storage (via, e.g., `Box<T>`) for each element in order to provide element references that just store pointers even while the container is allowed to grow, or might lazily move elements for which references are created to such indirected storage. This section’s proposed types are merely the simplest design points in a large space enabled by a flexible language mechanism.

5.3 Auto-Dereferencing for Syntactic Sugar

As one final ergonomic improvement, we note that by including the access path to the associated container, a deferred-borrow value contains all the information necessary to convert it to a true borrow automatically. The Rust language today contains a feature known as “auto-dereferencing” wherein the compiler inserts calls to the `deref()` or `deref_mut()` methods on smart pointer types when necessary. (This is similar to e.g. the use of operator overrides in C++ to implement smart pointers.) This allows transparent implementation of borrow/pointer-like values by library authors. We propose a modification to this desugaring step that, for a value `t` of type `T/a` that implements the `DefBorrow` or `DefBorrowMut` trait, invokes the deferred borrow `t.def_borrow(&a)` or `t.def_borrow_mut(&mut a)` as appropriate. This will make deferred-borrow references as ergonomic as true borrows in most circumstances, without additional user intervention.

5.4 Chained Deferred Borrows

We note that there is nothing preventing a deferred borrow’s path from referring to *another* deferred borrow object. In particular, consider the case where one vector container contains vectors as elements. A deferred borrow reference `p1` to an element of the outer vector `v1` might have type `Ref/v1`, and could in turn be used to produce a deferred borrow reference `p2`, which might have type `Ref/p1`. Auto-dereferencing could then chain *two* true borrows at the time of use, so that a write to `p2.a` becomes:

```
let v1: FrozenVec<FrozenVec<u32>> = ...;
let p1 = v1.deferred(0); // type FrozenVecRef/v1
let p2 = p1.deferred(0); // type FrozenVecRef/p1

// `p2.a = x` becomes:
let tmp1 = p1.def_borrow_mut(&mut v1); // type &mut FrozenVec<u32>, lifetime <: `v1`
let tmp2 = tmp1.def_borrow_mut(tmp1); // type &mut u32, lifetime <: `tmp1`
*tmp2 = x;
// tmp2 and tmp1 now out of scope; mutable borrow on `v1` ends.
```

6 Deferred-Borrow Prototype: Emulating Paths in Stable Rust

In order to evaluate the utility of deferred borrows, we implemented a prototype library of container types that provide deferred-borrow element references. Ideally, such a library would make use of true path-dependent types, as we described in §4. For expediency of implementation and experimentation, we instead chose to emulate path-dependent types with type-tagging, which is a strategy that works in stable Rust today (§6.1). We then illustrate several examples of our container library using this strategy (§6.2).

6.1 Emulating Path-Dependent Types with Type Tagging

Because a path-dependent type is a sort of dependent type – that is, because the type depends on a value in the program – we cannot implement our proposed system as written in today’s stable Rust language. Instead, we can *emulate* many of the type-safety benefits of path-dependent types, albeit without the convenience of auto-dereferences and with slightly more syntactic noise, by (i) *tagging* the container and its references with an extra type parameter, such that the “tag” type must match for a deference to work, and then (ii) using a *unique type for every container allocation site*. This strategy is less powerful than a true path-dependent type because it will let different objects from the same allocation site intermingle references, but is sufficient to understand the annotation burden and verify that the general approach can work.³

To understand this approach, consider first the following snippet using the “true” library design with path-dependent types:

```
fn main() {
    let v: AppendOnlyVec<T> = ...;
    let ref1: AppendOnlyVecRef<T>/v = v.deferred(i);
    let w: AppendOnlyVec<T> = ...;
    let ref2: AppendOnlyVecRef<T>/w = w.deferred(j);
    *ref1 = ...; // auto-derefs to: *ref1.def_borrow_mut(&mut v) = ...;

    // This is a type error (we used `w`, not `v`, but ref1 is of type `.../v`)
    // *ref1.def_borrow_mut(&mut w) = ...;
}
```

Instead, we define our container type with an extra type parameter `Tag`: hence, the container type becomes `AppendOnlyVec<T, Tag>`, and its deferred-borrow references are of type `AppendOnlyVecRef<T, Tag>`. Thus the example becomes:

```
fn main() {
    struct Tag1 {} // We can wrap this in a macro! (see below)
    let v: AppendOnlyVec<T, Tag1> = ...;
    let ref1: AppendOnlyVecRef<T, Tag1> = v.deferred(i);

    struct Tag2 {}
    let w: AppendOnlyVec<T, Tag2> = ...;
    let ref2: AppendOnlyVecRef<T, Tag2> = w.deferred(j);
    *ref1.def_borrow_mut(&mut v) = ...;
}
```

³ Note that this is subtly different than ownership-type approaches that encode ownership as a generic parameter, such as Potanin et al. [17]: while that work’s system enforces particular object instances as owners in a principled way, our prototype approach simply ties the path-dependent type to a *static allocation site*, which may produce many object instances. The only advantage of our scheme is that it can be written in Rust’s existing type system.

Note that this tag-type approach is *not* as strict as a true static path-dependent type, even as our example illustrates that the tag types can distinguish references from ‘v’ and ‘w’. Consider the case where a container is allocated within a loop: each instance, on each iteration, must have the same type, but logical memory safety requires disallowing a deferred-borrow from one to be used in a dereference with another.

However, this prototype has some value: it lets us see an approximation of the annotation burden, in that deferred-borrow types `Ref<T>/v` become `Ref<T, V>`. This is enough to evaluate the feasibility of large-program refactorings.

6.2 Macros for Tag Types

In order to make this strategy feasible and ergonomic enough for reasonable prototype use, we define several macros alongside our library of container types so that (i) container definition (with tag type) and (ii) deferred-reference access, which would become an invisible auto-deref with true path-dependent types, are both relatively simple.

First, we define a macro that defines a new empty tag type and parameterizes a container constructor:

```
fn main() {
  let v = vec![1,2,3,4];
  let mut v = freeze!(FrozenVec, v);
}
```

This expands to a struct-type definition inside a new scope (hence invisible to the rest of the program) and a constructor invocation parameterized on this tag type. The multiple instances that are produced by this expression are not distinguished by the type system as they would be with path-dependent types, but they are distinguished from other containers in the program that happen to coincide in the element type. (This is thus a form of allocation-site newtype idiom.)

Then, we define a macro that provides a short form for the deferred borrow itself, allowing the above dereferences to become simply:

```
fn ref<Tag>(v: &mut FrozenVec<u32, Tag>, elem: FrozenVecRef<u32, Tag>) {
  let value = *d!(v, elem);
  *dmut!(v, elem) += 1;
}
```

7 Qualitative Evaluation: Newly Possible Programming Patterns

We briefly describe several programming patterns that are possible with deferred borrows but not while restricted to true borrows, and discuss how API design considerations change when more flexible object references are possible.

7.1 Use Case #1: Graph Library

Consider the use-case of a general graph library: a top-level `Graph` object owns many `Node` instances, and each node contains out-edges to other nodes, with some `Edge` data attached to each out-edge. A straightforward Rust implementation, without deferred borrows, would simply hold `Nodes` in an array, and use indices to refer to them from other nodes:

30:20 Safe, Flexible Aliasing with Deferred Borrows

```
pub struct Graph {
    nodes: Vec<Node>,
}

pub type NodeIndex = usize;
pub struct Node {
    out_edges: Vec<(Edge, NodeIndex)>,
}

pub type EdgeIndex = usize;
```

We can then provide an API that allows for insertion of nodes and edges, and allows for accessing or mutating the data at each node or edge:

```
impl Graph {
    pub fn add_node(&mut self, node: Node) -> NodeIndex { ... }
    pub fn add_edge(&mut self, from: NodeIndex, to: NodeIndex, data: Edge) { ... }

    pub fn node<'a>(&'a self, node: NodeIndex) -> &'a Node { ... }
    pub fn node_mut<'a>(&'a mut self, node: NodeIndex) -> &'a mut Node { ... }
    pub fn edge<'a>(&'a self, node: NodeIndex, edge: EdgeIndex) -> &'a Edge { ... }
    pub fn edge_mut<'a>(&'a mut self, node: NodeIndex, edge: EdgeIndex)
        -> &'a mut Edge { ... }

    // ...
}
```

The two primary issues with such an API, as we have described already in motivating our approach, are (i) verbosity in use, due to the need to handle indices differently than native pointers/borrows, and (ii) logical unsafety because indices may be forged by the user, or erroneously taken from other contexts (e.g., another graph).

Thus, the following code is valid, but produces an unexpected result, because the programmer mixes indices from different domains (two different graphs) and erroneously uses an index in the wrong domain (here, a node index for graph `g1` used to index into `g2`'s nodes):

```
fn graphs_are_isomorphic(g1: &Graph, g2: &Graph) -> Mapping {
    'l: for mapping in generate_all_mappings() { // brute-force
        for n1 in g1.node_ids() {
            let n2 = mapping.map_node(n1);
            //          TYPO / logical error --v
            if !nodes_are_isomorphic(g1, g2, n1, n1, &mapping) {
                continue 'l;
            }
        }
        return mapping;
    }
}
```

In addition, accessing node data is cumbersome: each access must be written as `g.nodes[i]` rather than simply `p` (where `p` is a borrow). Note that the API user cannot simply take and save multiple borrows: if any borrow is live (mutable or immutable), no other mutable borrow can be created or used. (This may not be an issue for code that simply queries a data structure, though the lifetime annotations can still be difficult to manage. However it is surely an issue for any code that updates a data structure while holding multiple pointers into its inner structure.) As a result of this limitation, typical function bodies might look as follows:

```

fn nodes_are_isomorphic(g1: &Graph, g2: &Graph, n1: NodeIndex, n2: NodeIndex,
                        mapping: &Mapping) {
    if !data_is_equal(&g1.nodes[n1].data, &g2.nodes[n2].data) {
        return false;
    }
    // ...
}

fn mutate_nodes<F: Fn(&mut Node)>(g: &mut Graph, root: NodeIndex, mutate: F) {
    mutate(&mut g.nodes[root]);
    for i in 0..g.nodes[root].neighbor_count() {
        let neighbor = g.nodes[root].get_neighbor(i);
        if /* ... !visited(neighbor) ... */ {
            mutate_nodes(g, neighbor, mutate);
        }
    }
}

```

In contrast, a graph library that makes use of the deferred borrow pattern and the path-dependent types extension to Rust could define an API as follows:

```

impl Graph {
    pub fn add_node(&mut self, node: Node) -> NodeRef/0 { ... }
    // ...
}

impl DefBorrow<Node, Graph> for NodeRef {
    fn def_borrow<'a>(&self/a, g: &'a Graph) -> &'a Node { ... }
}

impl DefBorrowMut<Node, Graph> for NodeRef {
    fn def_borrow_mut<'a>(&mut self/a, g: &'a Graph) -> &'a mut Node { ... }
}

```

Given this API, the above functions could be rewritten as below, using generic path parameters (§4.4) to the Mapping object (definition omitted here) so that it can accept and produce indices associated with each graph object:

```

fn graphs_are_isomorphic(g1: &Graph, g2: &Graph) -> Mapping</0, /1> {
    'l: for mapping in generate_all_mappings() { // brute-force
        for n1 in g1.nodes() { // n1 is of type: NodeRef/g1
            let n2 = mapping.map_node(n1); // n2 is of type: NodeRef/g2
            // typecheck error caught statically -----v
            if !nodes_are_isomorphic(g1.node(n1), g2.node(n1), &mapping) {
                continue 'l;
            }
        }
        return mapping;
    }
}

```

Furthermore, if auto-dereference behavior is implemented for the deferred-borrow traits, then node accesses become much more convenient:

```

fn nodes_are_isomorphic(g1: &Graph, g2: &Graph, n1: NodeRef/0, n2: NodeRef/1,
                        mapping: &Mapping</0, /1>) {
    if !data_is_equal(&n1.data, &n2.data) {
        return false;
    }
    // ...
}

```

30:22 Safe, Flexible Aliasing with Deferred Borrows

```
fn mutate_nodes<F: Fn(&mut Node)>(g: &mut Graph, root: NodeRef/0, mutate: F) {
    mutate(&mut *root); // auto-deref.
    for i in 0..root.neighbor_count() {
        let neighbor = root.get_neighbor(i);
        if /* ... !visited(neighbor) ... */ {
            mutate_nodes(g, neighbor, mutate);
        }
    }
}
```

Note that the *efficiency* of these node accesses can be adjusted in a tradeoff with graph-mutation flexibility. If the user is willing to accept that the graph is frozen at a certain node count (many graph algorithms have this property: they do not mutate the graph topology, only data at each node/edge), then nodes can be stored in a `FrozenVec`, and a `NodeRef` is exactly as efficient as a true pointer, because it compiles down to exactly that. In contrast, if the programmer desires to allow graph expansion, an `AppendOnlyVec` could be used. If deletions are also desired, a more complex reference type might be used that dynamically prevents deletions of nodes with outstanding references. In short, the deferred-borrow idiom allows code to be *generic to the particular reference/addressing scheme*.

Finally, we note that the lessons we have learned from this example also apply to more heterogeneous or general data structures with arbitrary object-to-object linkage. For example, a tree with parent pointers (thus creating cycles between parent and children) or cross-links, could store tree nodes in an array and use deferred-borrow references throughout.

7.2 Use Case #2: Entity-Component Systems

Many programs operate on a large number of objects that fall into a small number of categories, and must hold references to these objects throughout their data structures. A common pattern is the *entity-component system*: the program has some global context with a few arrays or dynamically-sized vectors, one per object type; and a reference to an object of type `T` is simply an index into the vector of all `T`s. This provides efficiency advantages by allowing for more compact references (e.g., 32-bit indices instead of 64-bit pointers), more compact heap layout, and more efficient access patterns (i.e., streaming through an array in order rather than pointer-chasing). As such, the pattern is often used in high-performance scenarios such as game programming.

If we use deferred-borrow references for every entity in the system, we again have the guarantee against incorrect use of indices: e.g., an index into the array of all `T` objects cannot be used to index into the array of `U` objects instead. The path-dependent types are particularly ergonomic in this use pattern, however, because it is already the case that nearly every function will be passed a top-level “context” that allows access to the entity arrays; the reference types simply have types with paths starting at that context, and are thus automatically usable anywhere in the program without further plumbing. For example:

```
fn f(ctx: &mut Ctx, t: TRef/0, u: URef/0) {
    t.do_stuff(ctx);
    u.mutate(ctx);
    t.operate_with(ctx, u);
}
```

As a real-world test of this hypothesis, we took a small program, a microarchitectural CPU simulator, consisting of around 6000 lines of Rust. The simulator is written approximately in the style described above: references to major components of the simulated system (CPU

cores, caches, memory banks, etc.) are all by IDs that are indices into simulator-wide arrays. We adapted the system so that, instead, it would use deferred borrows to refer to CPU cores. The diff for this change (using the tag-type-based prototype library described in §6) was approximately 200 lines, almost all of which were in function signatures or field types within struct definitions.

7.3 Use Case #3: Logical Safety in an Array-based Algorithm

We note that static path-dependent types have uses outside of the deferred-borrow pattern. In fact, they are applicable as a means of tying references or handles to a particular context wherever such handles occur.

To see one such example, let us consider the case of an algorithm that operates on arrays of data, and manipulates indices into those arrays. It is often the case that such code is error-prone to write: the programmer might confuse which index corresponds to which array.

Let us say, for example, that we wish to develop an edit-distance algorithm that takes two strings (arrays of characters):

```
fn edit_distance(s1: &[u8], s2: &[u8]) -> usize {
  for i in 0..s1.len() {
    for j in 0..s2.len() {
      do_stuff(s1[i], s2[j]); // correct
      do_stuff(s1[i], s2[i]); // logical error!
    }
  }
}
```

One could make use of static path-dependent types to make such a logical error impossible, by defining a type `SafeSlice<T>` that wraps a `&[T]` and provides deferred-borrow-like element references:

```
fn edit_distance(s1: SafeSlice<u8>, s2: SafeSlice<u8>) -> usize {
  for i in s1.indices() {
    for j in s2.indices() {
      do_stuff(s1, s2, i, j);
      do_stuff(s1, s2, i, i); // caught at compile time!
    }
  }
}
```

In fact, path-dependent types are far more powerful than our examples have demonstrated so far: they serve, in brief, as a way to ensure *unforgeable values* that are produced and consumed by some opaque manager object and usable only with that object. We believe this type-system primitive would have many other use cases in a systems programming language such as Rust.

8 Related Work

Many prior works have explored the tradeoff space in type-system approaches to sound, usable memory-safety with useful aliasing guarantees that allow for parallelism. While we build on the particular programming language Rust in this work, Rust borrows from a long line of work on *ownership-based* and *region-based* memory-management. In addition, we adapt *path-dependent types*, which are a limited form of generalized dependent types, to provide type safety. To our knowledge, this is the first work to combine path-dependent types with a region-based ownership or borrowing system to provide static safety guarantees for a more flexible form of “borrowed” ownership.

Ownership tracking originated with Ownership Types [5] and spawned a long line of followup works [4, 3, 1, 6, 9, 15] to encode “contexts” or ownership domains, reason about split or fractional ownership and varying levels of access permission, allow “ombudsmen” that provide a safe external reference to internal state, and other approaches. In general, these past works begin with a highly restricted system – objects form a strict ownership tree, and access to an object is possible only by its owner – and then systematically relax that constraint to allow for common programming idioms to be expressed.

Rust’s borrowing system adapts *region-based memory management* ideas from Cyclone [8] to provide safe borrows of subtrees of the ownership tree. The borrows have constrained lifetimes, based on lexical regions, and the lifetimes of borrows of the same owned object are mutually disjoint. Abstractly, borrowing can also be seen as a form of permission system; effects and permissions have been widely studied as a means to provide memory safety and allow for deterministic parallel processing [9, 12].

Path-dependent types are a subset of dependent types [23], and have been extensively studied in the context of object-oriented languages, starting with Ernst [7] and Odersky et al. [14]. These works use path-dependent types to tie together related specific object instances, exactly as we use them to tie element references to the appropriate containers. In the context of the Scala language, path-dependent types have been formalized and serve as a foundational abstraction in the language [2], although Scala’s path-dependent types associate a new type with each value or class instance at runtime, and are hence a dynamic concept; the Scala implementation performs dynamic checks as a result. In contrast, our use of path-dependent types is purely static. Other sorts of dependent types, such as constraint types [13], can also encode restrictions on values which can in principle be used to build safe containers as we have. Such schemes are powerful and general; in contrast, our approach is specific to the problem of associating one object with another specific object.

There has been some work on how to build container data types in the context of ownership systems, or to fit a more regimented ownership system. Potanin et al. [16] modify standard Java container types to conform to an *owner-as-accessor* ownership system: this means that they must ensure that direct object accesses to internal state only go through the container itself. This is similar to how our deferred-borrow objects contain state that allows a container to return a direct reference at time of use.

9 Future Work and Conclusion

In this paper, we have examined the shortcomings of *ownership and borrow-based memory management* as practiced in the Rust programming language. Although borrows with static lifetimes allow the compiler to verify that (i) no dangling pointers exist, and (ii) no aliasing mutable pointers exist, the imprecision in borrow tracking that arises when *container data types* are used frequently creates friction for programmers. A standard idiom is to refer to container elements by index, manually indexing an element each time a short-lived borrow is required. However, this approach is cumbersome and is *logically* unsafe because an index is not tied to the container. We introduce *deferred borrows*, which encapsulate a memory-safe reference to an element of a container. A deferred borrow provides the *same static guarantees* as an ordinary borrow: it will never be dangling and it will never allow aliasing mutable pointers to exist. This is achieved by performing a true borrow of the associated container only when the deferred borrow is actually used, rather than when it is created. The associated container is tied to the deferred borrow with the use of *static path-dependent types*.

While we have presented a complete proposal, we believe there are several angles for further expansion of this language feature that are promising. First, while the deferred-borrow implementation is currently manual (the container type must return a value with a type that implements the `DefBorrow` trait), it could be auto-derived from an ordinary access method that returns a true borrow. The compiler should be able to analyze the method body to determine whether actions are safe to defer based on just a few assumptions (e.g., that the internal storage of a `Vec` will not be reallocated).

Second, we have not discussed *field* borrows so far, but conceptually a field accessor (`x.a`) is a deferred-borrow operator, or selector, whose state happens to be constant. The Rust borrow checker obviates the need for deferred field borrows in many common cases because it can directly track individual fields (it natively understands field access paths), but reformulating field accesses in terms of deferred borrows may provide an opportunity to simplify the borrow checker.

Finally, deferred borrows, when seen as operators (curried with specific state) that convert the root container borrow to an element borrow at the time of use, should be *composable* as well. This is especially useful when considering field accessors as deferred-borrow state. In essence, deferred borrows encapsulate an arbitrary access path, possibly constructed by concatenating access path components, allowing the program to refer to state without restricting access to portions of the state tree in the meantime.

We believe that the deferred-borrow approach will be a valuable addition to the repertoire of safe memory management techniques in Rust and related ownership-and-borrowing-based type systems in the future.

References

- 1 J Aldrich and C Chambers. Ownership domains: Separating aliasing policy from mechanism. *ECOOP*, 2004.
- 2 N Amin, T Rompf, and M Odersky. Foundations of path-dependent types. *OOPSLA*, 2014.
- 3 C Boyapati, A Sălcianu, W Beebe Jr., and M Rinard. Ownership types for safe region-based memory management in real-time Java. *PLDI*, 2003.
- 4 D Clarke and S Drossopoulou. Ownership, encapsulation and the disjointness of type and effect. *OOPSLA*, 2002.
- 5 D G Clarke, J M Potter, and J Noble. Ownership types for flexible alias protection. *OOPSLA*, 1998.
- 6 W Dietl, S Drossopoulou, and P Müller. Generic universe types. *ECOOP*, 2007.
- 7 E Ernst. Family polymorphism. *ECOOP*, 2001.
- 8 D Grossman, G Morrisett, T Jim, M Hicks, Y Wang, and J Cheney. Region-based memory management in Cyclone. *PLDI*, 2002.
- 9 R L Bocchino Jr., V S Adve, D Dig, S V Adve, S Heumann, R Komuravelli, J Overbey, P Simmons, H Sung, and M Vakilian. A type and effect system for Deterministic Parallel Java. *OOPSLA*, 2009.
- 10 G. Kastrinis, G. Balatsouras, K. Ferles, N. Prokopaki-Kostopoulou, and Y. Smaragdakis. An efficient data structure for must-alias analysis. *CC*, 2018.
- 11 K R M Leino, A Poetzsch-Heffter, and Y Zhou. Using data groups to specify and check side effects. *PLDI*, 2002.
- 12 K Naden, R Bocchino, J Aldrich, and K Bierhoff. A type system for borrowing permissions. *POPL*, 2011.
- 13 N Nystrom, V Saraswat, J Palsberg, and C Grothoff. Constrained types for object-oriented languages. *OOPSLA*, 2008.
- 14 M Odersky, V Cremet, C Röckl, and M Zenger. A nominal theory of objects with dependent types. *ECOOP*, 2003.

30:26 Safe, Flexible Aliasing with Deferred Borrows

- 15 J Östlund and T Wrigstad. Multiple aggregate entry points for ownership types. *ECOOP*, 2012.
- 16 A Potanin, M Damitio, and J Noble. Are your incoming aliases really necessary? counting the cost of object ownership. *ICSE*, 2013.
- 17 A Potanin, J Noble, D Clarke, and R Biddle. Generic ownership for generic java. *OOPSLA*, 2006.
- 18 Rust community. The Rust programming language. <https://www.rust-lang.org/>.
- 19 Rust community. Rust survey 2018 results. <https://blog.rust-lang.org/2018/11/27/Rust-survey-2018.html>.
- 20 S Stork, K Naden, J Sunshine, M Mohr, A Fonseca, P Marques, and J Aldrich. Aeminium: A permission-based concurrent-by-default programming language approach. *ACM Trans. Prog. Lang. Sys.*, 36, March 2014.
- 21 A Weiss, D Patterson, ND Matsakis, and A Ahmed. Oxide: the essence of Rust, 2019. [arXiv:1903.00982](https://arxiv.org/abs/1903.00982).
- 22 E Westbrook, J Zhao, Z Budimlić, and V Sarkar. Practical permissions for race-free parallelism. *ECOOP*, 2012.
- 23 Hongwei Xi and Frank Pfenning. Dependent types in practical programming. *POPL*, 1999.

Reshape Your Layouts, Not Your Programs: A Safe Language Extension for Better Cache Locality

Alexandros Tasos

Imperial College London, United Kingdom
at1917@ic.ac.uk

Juliana Franco

Microsoft Research, London, United Kingdom
juliana.franco@microsoft.com

Sophia Drossopoulou

Imperial College London, United Kingdom
Microsoft Research, London, United Kingdom
scd@doc.ic.ac.uk

Tobias Wrigstad

Uppsala University, Sweden
tobias.wrigstad@it.uu.se

Susan Eisenbach

Imperial College London, United Kingdom
sue@doc.ic.ac.uk

Abstract

The vast gap between CPU and RAM speed means that on modern architectures, developers need to carefully consider data placement in memory to exploit spatial and temporal cache locality and use CPU caches effectively. To that extent, developers have devised various strategies regarding data placement; for objects that should be close in memory, a contiguous *pool* of objects is allocated and then new instances are constructed inside it; an array of objects is *clustered* into multiple arrays, each holding the values of a specific field of the objects¹. Such data placements, however, have to be performed manually, hence readability, maintainability, memory safety, and key OO concepts such as encapsulation and object identity need to be sacrificed and the business logic needs to be modified accordingly.

We propose a language extension, **SHAPES**, which aims to offer developers high-level fine-grained control over data placement, whilst retaining memory safety and the look-and-feel of OO. **SHAPES** extends an OO language with the concepts of *pools* and *layouts*: Developers declare *pools* that contain objects of a specific type and specify the pool's *layout*. A *layout* specifies how objects in a pool are laid out in memory. That is, it dictates how the values of the fields of the pool's objects are grouped together into *clusters*. Objects stored in pools behave identically to ordinary, standalone objects; the type system allows the code to be oblivious to the layout being used. This means that the business logic is completely decoupled from any placement concerns and the developer need not deviate from the spirit of OO to better utilise the cache.

In this paper, we present the features of **SHAPES**, as well as the design rationale behind each feature. We then showcase the merit of **SHAPES** through a sequence of case studies; we claim that, compared to the manual pooling and clustering of objects, we can observe improvement in readability and maintainability, and comparable (i.e., on par or better) performance.

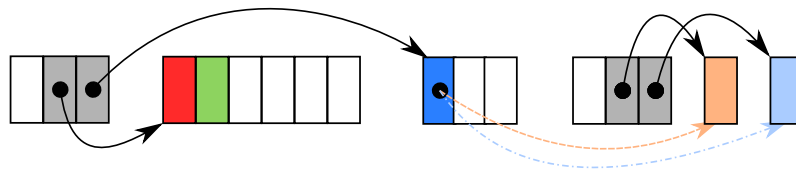
We also present **SHAPES^h**, an OO calculus which models the **SHAPES** ideas, we formalise the type system, and prove soundness. The **SHAPES^h** type system uses ideas from Ownership Types [1] and Java Generics [2]: In **SHAPES^h**, pools are part of the types; **SHAPES^h** class and type definitions are enriched with pool parameters. Moreover, class pool parameters are enriched with bounds, which

¹ Commonly referred to as an *Array-of-Structs* (AoS) to *Struct-of-Arrays* (SoA) transformation.



31:2 Reshape Your Layouts, Not Your Programs

```
1 class Professor<pProf: [Professor<pProf>]> {
2     name: String; ssn: String;
3 }
4 class Student<pStu: [Student<pStu, pProf>], pProf: [Professor<pProf>]> {
5     name: String; age: int; supervisor: Professor<pProf>;
6 }
7 layout ProfL: Professor = rec{name} + rec{ssn};
8 layout StuL: Student = rec{name, age} + rec{supervisor};
9 ...
10 pools pStu1: StuL<pStu1, pProf1>, pProf1: ProfL<pProf1>;
11 stu = new Student<pStu1, pProf1>;
12 prof = new Professor<pProf1>;
13 stu.supervisor = prof;
```



■ **Figure 1** Example SHAPES code and memory layout.

is what allows the business logic of SHAPES to be oblivious to the layout being used. SHAPES^h types also enforce pool uniformity and homogeneity. A pool is uniform if it contains objects of the same class only; a pool is homogeneous if the corresponding fields of all its objects point to objects in the same pool. These properties allow for more efficient implementation.

For performance considerations, we also designed SHAPES^l, an untyped, unsafe low-level language with no explicit support for objects or pools. We argue that it is possible to translate SHAPES^l into existing low-level intermediate representations, such as LLVM [3], present the translation of SHAPES^h into SHAPES^l, and show its soundness.

Thus, we expect SHAPES to offer developers more fine-grained control over data placement, without sacrificing memory safety or the OO look-and-feel.

2012 ACM Subject Classification Software and its engineering → Classes and objects; Theory of computation → Formalisms; General and reference → Performance

Keywords and phrases Cache utilisation, Data representation, Memory safety

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2020.31

Category SCICO Journal-first

Related Version Full article available at <https://doi.org/10.1016/j.scico.2020.102481>.

Supplementary Material ECOOP 2020 Artifact Evaluation approved artifact available at <https://doi.org/10.4230/DARTS.6.2.19>.

Funding *Alexandros Tasos*: Supported by an EPSRC Centre for Doctoral Training in High Performance Embedded and Distributed Systems (HiPEDS) Grant (Reference EP/L016796/1).

References

- 1 Dave Clarke, Johan Östlund, Ilya Sergey, and Tobias Wrigstad. *Ownership Types: A Survey*, pages 15–58. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013. doi:10.1007/978-3-642-36946-9_3.
- 2 James Gosling, Bill Joy, Guy Steele, Gilad Bracha, and Alex Buckley. *The Java Language Specification*, Java SE 8 Edition (Java Series), 2014.
- 3 Chris Lattner and Vikram Adve. Lvm: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization*, page 75. IEEE Computer Society, 2004.

A Big Step from Finite to Infinite Computations

Davide Ancona 


DIBRIS, University of Genova, Italy
davide.ancona@unige.it

Francesco Dagnino 

DIBRIS, University of Genova, Italy
francesco.dagnino@dibris.unige.it

Jurriaan Rot

Radboud University, The Netherlands
jrot@cs.ru.nl

Elena Zucca 

DIBRIS, University of Genova, Italy
elena.zucca@unige.it

Abstract

The known is finite, the unknown infinite
– Thomas Henry Huxley

The behaviour of programs can be described by the final *results* of computations, and/or their interactions with the context, also seen as *observations*. For instance, a function call can terminate and return a value, as well as have output effects during its execution.

Here, we deal with semantic definitions covering both results and observations. Often, such definitions are provided for *finite* computations only. Notably, in big-step style, infinite computations are simply not modelled, hence diverging and stuck terms are not distinguished. This becomes even more unsatisfactory if we have observations, since a non-terminating program may have significant infinite behaviour.

Recently, examples of big-step semantics modeling divergence have been provided [3, 4] by means of *generalized inference systems* [2, 5], which allow *corules* to control coinduction. Indeed, modeling infinite behaviour by a purely coinductive interpretation of big-step rules would lead to spurious results [6] and undetermined observation, whereas, by adding appropriate corules, we can correctly get divergence (∞) as the only result, and a uniquely determined observation. This approach has been adopted in [3, 4] to design big-step definitions including infinite behaviour for lambda-calculus and a simple imperative Java-like language. However, in such works the designer of the semantics is in charge of finding the appropriate corules, and this is a non-trivial task.

In this paper, we show a general construction that extends a given big-step semantics, modeling finite computations, to include infinite behaviour as well, notably by generating appropriate corules. The construction consists of two steps:

1. Starting from a monoid O modeling finite observations (e.g., finite traces), we construct an ω -monoid (O, O_∞) also modeling infinite observations (e.g., infinite traces). The latter structure is a variation of the notion of ω -semigroup [7], including a *mixed product* composing a finite with a possibly infinite observation, and an *infinite product* mapping an infinite sequence of finite observations into a single one (possibly infinite).
2. Starting from an inference system defining a big-step judgment $c \Rightarrow \langle r, o \rangle$, with c denoting a configuration, $r \in R$ a result, and $o \in O$ a finite observation, we construct an inference system with corules defining an extended big-step judgment $c \Rightarrow \langle r_\infty, o_\infty \rangle$ with $r_\infty \in R_\infty = R + \{\infty\}$, and $o_\infty \in O_\infty$ a “possibly infinite” observation. The construction generates additional rules for *propagating divergence*, and corules for *introducing divergence* in a controlled way.

The exact corules added in the construction depend on the type of observations that one starts with. To show the effectiveness of our approach, we provide several instances of the framework, with different kinds of (finite) observations.



© Davide Ancona, Francesco Dagnino, Jurriaan Rot, and Elena Zucca;
licensed under Creative Commons License CC-BY

34th European Conference on Object-Oriented Programming (ECOOP 2020).

Editors: Robert Hirschfeld and Tobias Pape; Article No. 32; pp. 32:1–32:2

Leibniz International Proceedings in Informatics



LIPIC Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

32:2 A Big Step from Finite to Infinite Computations

Finally, we prove a correctness result for the construction. To this end, we assume the original big-step semantics to be equivalent to (finite sequences of steps in) a reference small-step semantics, and we show that, by applying the construction, we obtain an extended big-step semantics which is still equivalent to the small-step semantics, where we consider possibly infinite sequences of steps. As hypotheses, rather than just equivalence in the finite case (which would be not enough), we assume a set of *equivalence conditions* between individual big-step rules and the small-step relation. This proof of equivalence holds for deterministic semantics; issues arising in the non-deterministic case and a possible solution are sketched in the conclusion of the full paper.

2012 ACM Subject Classification Theory of computation → Operational semantics; Software and its engineering → Recursion; Software and its engineering → Semantics

Keywords and phrases Operational semantics, coinduction, infinite behaviour

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2020.32

Category SCICO Journal-first

Related Version A full version of the paper is available at [1], <https://doi.org/10.1016/j.scico.2020.102492>.

Funding *Davide Ancona*: Member of GNCS (Gruppo Nazionale per il Calcolo Scientifico), INdAM (Istituto Nazionale di Alta Matematica “F. Severi”)

References

- 1 Davide Ancona, Francesco Dagnino, Jurriaan Rot, and Elena Zucca. A big step from finite to infinite computations. *Science of Computer Programming*, page 102492, 2020. doi:10.1016/j.scico.2020.102492.
- 2 Davide Ancona, Francesco Dagnino, and Elena Zucca. Generalizing inference systems by coaxioms. In Hongseok Yang, editor, *ESOP 2017 - European Symposium on Programming*, volume 10201 of *Lecture Notes in Computer Science*, pages 29–55. Springer, 2017. doi:10.1007/978-3-662-54434-1_2.
- 3 Davide Ancona, Francesco Dagnino, and Elena Zucca. Reasoning on divergent computations with coaxioms. *PACMPL*, 1(OOPSLA):81:1–81:26, 2017. doi:10.1145/3133905.
- 4 Davide Ancona, Francesco Dagnino, and Elena Zucca. Modeling infinite behaviour by corules. In Todd D. Millstein, editor, *ECOOP’18 - Object-Oriented Programming*, volume 109 of *LIPIcs*, pages 21:1–21:31. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2018. doi:10.4230/LIPIcs.ECOOP.2018.21.
- 5 Francesco Dagnino. Coaxioms: flexible coinductive definitions by inference systems. *Logical Methods in Computer Science*, 15(1), 2019. doi:10.23638/LMCS-15(1:26)2019.
- 6 Xavier Leroy and Hervé Grall. Coinductive big-step operational semantics. *Information and Computation*, 207(2):284–304, 2009. doi:10.1016/j.ic.2007.12.004.
- 7 Dominique Perrin and Jean-Eric Pin. *Infinite words - automata, semigroups, logic and games*, volume 141 of *Pure and applied mathematics series*. Elsevier Morgan Kaufmann, 2004.

Abstracting Gradual References

Matías Toro

PLEIAD Laboratory, Computer Science Department (DCC), University of Chile, Santiago, Chile
mtoro@dcc.uchile.cl

Éric Tanter

PLEIAD Laboratory, Computer Science Department (DCC), University of Chile, Santiago, Chile
etanter@dcc.uchile.cl

Abstract

Gradual typing is an effective approach to integrate static and dynamic typing, which supports the smooth transition between both extremes via the (programmer-controlled) precision of type annotations [19, 21]. Imprecision is normally introduced via the unknown type $?$, e.g. function type $\text{Int} \rightarrow \text{Bool}$ is more precise than $? \rightarrow ?$, and both more precise than $?$. Gradual typing relates types of different precision using consistent type relations, such as *type consistency* (resp. *consistent subtyping*), the gradual counterpart of type equality (resp. subtyping). For instance, $? \rightarrow \text{Int}$ is consistent with $\text{Bool} \rightarrow ?$. This approach has been applied in a number of settings, such as objects [20], subtyping [20, 11], effects [4, 5], ownership [18], tpestates [27, 12], information-flow typing [9, 10, 23], session types [14], refinements [17], set-theoretic types [6], Hoare logic [3], parametric polymorphism [1, 2, 16, 15, 28, 24], and references [19, 13, 22].

In particular, gradual typing for mutable references has seen the elaboration of various possible semantics: *invariant* references [19], *guarded* references [13], *monotonic* references [22], and *permissive* references [22]. Invariant references are a form of references where reference types are invariant with respect to type consistency. Guarded references admit variance thanks to systematic runtime checks on reference reads and writes; the runtime type of an allocated cell never changes during execution. Guarded references have been formulated in a space-efficient coercion calculus, which ensures that gradual programs do not accumulate unbounded pending checks during execution. Hereafter, we refer to this language as HCC. Monotonic references favor efficiency over flexibility by only allowing reference cells to vary towards more precise types. This allows reference operations in statically-typed regions to safely proceed without any runtime checks. Permissive references are the most flexible approach, in which reference cells can be initialized and updated to any value of any type at any time.

These four developments reflect different design decisions with respect to gradual references: is the reference type constructor variant under consistency? Can the programmer specify a precise bound on the static type of a reference, and hence on the corresponding heap cell type? Can the heap cell type evolve its precision at runtime, and if yes, how? There is obviously no absolute answer to these questions, as they reflect different tradeoffs such as in efficiency and precision. This work explores the semantics that results from the application of a *systematic* methodology to gradualize static type systems. Currently we can find in the literature two methodologies to gradualize statically-typed languages: Abstracting Gradual Typing (AGT) [11], and the Gradualizer [7]. In this work, we consider the AGT methodology as it naturally scales to auxiliary structures such as a mutable heap.

The AGT methodology helps to systematically construct gradually-typed languages by using abstract interpretation [8] at the type level. In brief, AGT interprets gradual types as an abstraction of sets of possible static types, formally captured through a Galois connection. The static semantics of a gradual language are then derived by lifting the semantics of a statically-typed language through this connection, and the dynamic semantics follow by Curry-Howard from proof normalization of the type safety argument. The AGT methodology has been shown to be effective in many contexts: records and subtyping [11], type-and-effects [4, 5], refinement types [17, 26], set-theoretic and union types [6, 25], information-flow typing [23], and parametric polymorphism [24]. However, this methodology has never been applied to mutable references in isolation. Although Toro *et al.* [23] apply AGT to a language with references, they only gradualize *security levels* of types (e.g. $\text{Ref Int}_?$), not whole types (e.g. $\text{Ref } ?$ is not supported). In this article we answer the following open questions:



© Matías Toro and Éric Tanter;

licensed under Creative Commons License CC-BY

34th European Conference on Object-Oriented Programming (ECOOP 2020).

Editors: Robert Hirschfeld and Tobias Pape; Article No. 33; pp. 33:1–33:4

Leibniz International Proceedings in Informatics



LIPIC Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Which semantics for gradually-type references follows by systematically applying AGT? Does AGT justify one of the existing approaches, or does it suggest yet another design? Can we recover other semantics for gradual references, if yes, how?

This article first reviews the different existing gradual approaches to mutable references through examples. It then presents the semantics for gradual references that is obtained by applying AGT, and how to accommodate the other semantics. More specifically, this work makes the following contributions:

- We present λ_{REF} , a gradual language with support for mutable references. We derive λ_{REF} by applying the AGT methodology to a fully-static simple language with mutable references called λ_{REF} . This is the first application of AGT that focuses on gradually-typed mutable references.
- We prove that λ_{REF} satisfies the gradual guarantee of Siek *et al.* [21]. We also present the first formal statement and proof of the conservative extension of the dynamic semantics of the static language [21], for a gradual language derived using AGT.
- We prove that the derived language, λ_{REF} , corresponds to the semantics of guarded references from HCC. Formally, given a λ_{REF} term and its compilation to HCC⁺ (an adapted version of HCC extended with conditionals and binary operations) we prove that both terms are bisimilar, and that consequently they either both terminate, both fail, or both diverge.
- We observe that λ_{REF} and HCC⁺ differ in the order of combination of runtime checks. As a result, HCC is space efficient whereas λ_{REF} is not: we can write programs in λ_{REF} that may accumulate an unbounded number of checks. We formalize the changes needed in the dynamic semantics of λ_{REF} to achieve space efficiency. This technique to recover space efficiency is in fact independent from mutable references, and is therefore applicable to other gradual languages derived with AGT.
- We formally describe how to support other gradual reference semantics in λ_{REF} by presenting $\lambda_{\text{REF}}^{\text{pm}}$, an extension that additionally supports both *permissive* and *monotonic* references. Finally, we prove for the first time that monotonic references satisfy the dynamic gradual guarantee, a non-trivial result that requires careful consideration of updates to the store.

Additionally, we implemented λ_{REF} as an interactive prototype that displays both typing derivations and reduction traces. All the examples mentioned in this paper are readily available in the online prototype available at <https://pleiad.cl/grefs>.

As a result, this paper sheds further light on the design space of gradual languages with mutable references and contributes to deepening the understanding of the AGT methodology.

2012 ACM Subject Classification Theory of computation → Type structures; Theory of computation → Program semantics

Keywords and phrases Gradual Typing, Mutable References, Abstract interpretation

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2020.33

Category SCICO Journal-first

Related Version A full version of the paper is available at <https://doi.org/10.1016/j.scico.2020.102496>.

Funding *Matías Toro*: Partially funded by FONDECYT Project 3200583.

Éric Tanter: Partially funded by FONDECYT Project 1190058.

References

- 1 Amal Ahmed, Robert Bruce Findler, Jeremy G. Siek, and Philip Wadler. Blame for all. In *Proceedings of the 38th annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2011)*, pages 201–214, Austin, Texas, USA, January 2011. ACM Press.

- 2 Amal Ahmed, Dustin Jamner, Jeremy G. Siek, and Philip Wadler. Theorems for free for free: Parametricity, with and without types. *Proceedings of the ACM on Programming Languages*, 1(ICFP):39:1–39:28, September 2017.
- 3 Johannes Bader, Jonathan Aldrich, and Éric Tanter. Gradual program verification. In Işıl Dillig and Jens Palsberg, editors, *Proceedings of the 19th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI 2018)*, volume 10747 of *Lecture Notes in Computer Science*, pages 25–46, Los Angeles, CA, USA, January 2018. Springer-Verlag.
- 4 Felipe Bañados Schwerter, Ronald Garcia, and Éric Tanter. A theory of gradual effect systems. In *Proceedings of the 19th ACM SIGPLAN Conference on Functional Programming (ICFP 2014)*, pages 283–295, Gothenburg, Sweden, September 2014. ACM Press.
- 5 Felipe Bañados Schwerter, Ronald Garcia, and Éric Tanter. Gradual type-and-effect systems. *Journal of Functional Programming*, 26:19:1–19:69, September 2016.
- 6 Giuseppe Castagna and Victor Lanvin. Gradual typing with union and intersection types. *Proceedings of the ACM on Programming Languages*, 1(ICFP):41:1–41:28, September 2017.
- 7 Matteo Cimini and Jeremy Siek. The gradualizer: a methodology and algorithm for generating gradual type systems. In *Proceedings of the 43rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2016)*, pages 443–455, St Petersburg, FL, USA, January 2016. ACM Press.
- 8 Patrick Cousot and Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the 4th ACM Symposium on Principles of Programming Languages (POPL 77)*, pages 238–252, Los Angeles, CA, USA, January 1977. ACM Press.
- 9 Tim Disney and Cormac Flanagan. Gradual information flow typing. In *International Workshop on Scripts to Programs*, 2011.
- 10 Luminous Fennell and Peter Thiemann. Gradual security typing with references. In *Proceedings of the 26th Computer Security Foundations Symposium (CSF)*, pages 224–239, June 2013.
- 11 Ronald Garcia, Alison M. Clark, and Éric Tanter. Abstracting gradual typing. In *Proceedings of the 43rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2016)*, pages 429–442, St Petersburg, FL, USA, January 2016. ACM Press.
- 12 Ronald Garcia, Éric Tanter, Roger Wolff, and Jonathan Aldrich. Foundations of typestate-oriented programming. *ACM Transactions on Programming Languages and Systems*, 36(4):12:1–12:44, October 2014.
- 13 David Herman, Aaron Tomb, and Cormac Flanagan. Space-efficient gradual typing. *Higher-Order and Sympolic Computation*, 23(2):167–189, June 2010.
- 14 Atsushi Igarashi, Peter Thiemann, Vasco T. Vasconcelos, and Philip Wadler. Gradual session types. *Proceedings of the ACM on Programming Languages*, 1(ICFP):38:1–38:28, September 2017.
- 15 Yuu Igarashi, Taro Sekiyama, and Atsushi Igarashi. On polymorphic gradual typing. *Proceedings of the ACM on Programming Languages*, 1(ICFP):40:1–40:29, September 2017.
- 16 Lintaro Ina and Atsushi Igarashi. Gradual typing for generics. In *Proceedings of the 26th ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA 2011)*, pages 609–624, Portland, Oregon, USA, October 2011. ACM Press.
- 17 Nico Lehmann and Éric Tanter. Gradual refinement types. In *Proceedings of the 44th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2017)*, pages 775–788, Paris, France, January 2017. ACM Press.
- 18 Ilya Sergey and Dave Clarke. Gradual ownership types. In Helmut Seidl, editor, *Proceedings of the 21st European Symposium on Programming Languages and Systems (ESOP 2012)*, volume 7211 of *Lecture Notes in Computer Science*, pages 579–599, Tallinn, Estonia, 2012. Springer-Verlag.
- 19 Jeremy Siek and Walid Taha. Gradual typing for functional languages. In *Proceedings of the Scheme and Functional Programming Workshop*, pages 81–92, September 2006.

- 20 Jeremy Siek and Walid Taha. Gradual typing for objects. In Erik Ernst, editor, *Proceedings of the 21st European Conference on Object-oriented Programming (ECOOP 2007)*, number 4609 in Lecture Notes in Computer Science, pages 2–27, Berlin, Germany, July 2007. Springer-Verlag.
- 21 Jeremy G. Siek, Michael M. Vitousek, Matteo Cimini, and John Tang Boyland. Refined criteria for gradual typing. In *1st Summit on Advances in Programming Languages (SNAPL 2015)*, volume 32 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 274–293, Asilomar, California, USA, May 2015. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.
- 22 Jeremy G. Siek, Michael M. Vitousek, Matteo Cimini, Sam Tobin-Hochstadt, and Ronald Garcia. Monotonic references for efficient gradual typing. In Jan Vitek, editor, *Proceedings of the 24th European Symposium on Programming Languages and Systems (ESOP 2015)*, volume 9032 of *Lecture Notes in Computer Science*, pages 432–456, London, UK, March 2015. Springer-Verlag.
- 23 Matías Toro, Ronald Garcia, and Éric Tanter. Type-driven gradual security with references. *ACM Transactions on Programming Languages and Systems*, 40(4):16:1–16:55, November 2018.
- 24 Matías Toro, Elizabeth Labrada, and Éric Tanter. Gradual parametricity, revisited. *Proceedings of the ACM on Programming Languages*, 3(POPL):17:1–17:30, January 2019.
- 25 Matías Toro and Éric Tanter. A gradual interpretation of union types. In *Proceedings of the 24th Static Analysis Symposium (SAS 2017)*, volume 10422 of *Lecture Notes in Computer Science*, pages 382–404, New York City, NY, USA, August 2017. Springer-Verlag.
- 26 Niki Vazou, Éric Tanter, and David Van Horn. Gradual liquid type inference. *Proceedings of the ACM on Programming Languages*, 2(OOPSLA), November 2018.
- 27 Roger Wolff, Ronald Garcia, Éric Tanter, and Jonathan Aldrich. Gradual tpestate. In Mira Mezini, editor, *Proceedings of the 25th European Conference on Object-oriented Programming (ECOOP 2011)*, volume 6813 of *Lecture Notes in Computer Science*, pages 459–483, Lancaster, UK, July 2011. Springer-Verlag.
- 28 Ningning Xie, Xuan Bi, and Bruno C. d. S. Oliveira. Consistent subtyping for all. In Amal Ahmed, editor, *Proceedings of the 27th European Symposium on Programming Languages and Systems (ESOP 2018)*, volume 10801 of *Lecture Notes in Computer Science*, pages 3–30, Thessaloniki, Greece, April 2018. Springer-Verlag.